

USD Core Specification

v1.0.1 (b7bc21a)

AOUSD Core Spec Working Group Authors

2025-12-12



AOUSD
Alliance for OpenUSD

Contents

1	Copyright License Agreement	4
2	Foreword	5
2.1	Domain Specific Specifications	5
2.2	Document Conventions	5
2.3	Supplemental Materials	6
2.4	Versioning	6
2.5	Errata and Support	6
2.6	Changelog	7
3	Introduction	8
3.1	Representation and Authoring	8
3.2	Scene Construction and Composition	8
3.3	Specification Structure	8
4	Compliance	10
4.1	Interpreting Compliance	10
4.2	Compliance Ambiguities	10
4.3	Compliance Disagreements	10
4.4	Compliance Rubric	10
5	Glossary	13
5.1	Core Terms	13
5.2	Externally Defined Terms and Definitions	17
6	Foundational Data Types	20
6.1	Background	20
6.2	Scalar Types	20
6.3	Dimensioned Types	21
6.4	Algebraic Types	22
6.5	Semantic Aliases For Types	22
6.6	Container Types	23
6.7	References	31
7	Document Data Model	32
7.1	Scope	32
7.2	Layer Structure	32
7.3	Specs	32
7.4	Metadata Fields	35
7.5	Formats	41
7.6	Core Metadata Fields	43
7.7	References	59
8	Paths	60
8.1	Introduction	60
8.2	Element Ordering	63
8.3	Path Grammar	64
8.4	Conventions	65
8.5	Grammar Definition	65
8.6	Examples	70
8.7	Compatibility with Legacy Content	71
8.8	Runtime Considerations	74

8.9	References	74
9	Resource Interface	75
9.1	Scope	75
9.2	Resource Identifiers	75
9.3	Resource Protocols	76
9.4	Relative Resource Identifiers	76
9.5	Resolving Identifiers to Locations	77
9.6	Resolving Extensions	77
9.7	Packaged Resources	78
9.8	Scheme Specifications	78
9.9	Additional Notes	78
9.10	Security Considerations	79
9.11	References	79
10	Composition	80
10.1	Scope	80
10.2	Overview	80
10.3	Composition Operators	81
10.4	Strength Ordering	94
10.5	Namespace Mappings	101
10.6	Composition Errors	102
11	Stage Population	103
11.1	Scope	103
11.2	The Stage	103
11.3	Populating the Stage	103
11.4	Scene Graph Model Hierarchy	107
11.5	Stage Queries	108
12	Value Resolution	109
12.1	Scope	109
12.2	Metadata Resolution	109
12.3	Attribute Resolution	112
12.4	Relationships and Attribute Connections	133
12.5	Interpolation	134
13	Schemas	146
13.1	Scope	146
13.2	Extension Metadata Fields	146
13.3	Schema Types	147
13.4	Core Schema Types	157
14	Color	158
14.1	Supported Color Spaces	158
14.2	Core Metadata Extensions	159
14.3	ColorSpaceDefinitionAPI	160
14.4	ColorSpaceAPI	161
15	Collections	165
15.1	CollectionAPI	165
15.2	Authoring and Evaluating Collections	166
16	Core File Formats	169
16.1	Compatibility with Document Versions	169

16.2Text	169
16.3Binary	220
16.4Package	248
16.5References	250
17Closing	251
17.1Acknowledgements	251

1 Copyright License Agreement

Copyright © 2025 Alliance for OpenUSD (“AOUSD”)

This AOUSD Final Deliverable is made available under the terms of the Creative Commons Attribution-NoDerivatives 4.0 International License (CC BY-ND 4.0). As stated in the license, you may copy and redistribute this deliverable, provided that attribution is given to the Alliance for OpenUSD. However, if you create a derivative work from the deliverable, you may not distribute that work. A copy of the license is available at <https://creativecommons.org/licenses/by-nd/4.0/>.

Elements of this AOUSD Final Deliverable may be subject to third party intellectual property rights, including without limitation, patent, copyright or trademark rights, and any such third party may or may not be a member of the AOUSD. AOUSD members grant certain intellectual property rights, including patent rights, as set forth in the [AOUSD Working Group Charter for the Working Group that developed this AOUSD Final Deliverable](#). AOUSD is not responsible for, and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS AOUSD FINAL DELIVERABLE IS PROVIDED “AS IS.” AOUSD, its members and contributors expressly disclaim any warranties (express, implied, or otherwise), including implied warranties of merchantability, non-infringement, fitness for a particular purpose, or title, related to the AOUSD Final Deliverable. The entire risk as to implementing or otherwise using this AOUSD Final Deliverable is assumed by the implementer and user. IN NO EVENT WILL AOUSD, ITS MEMBERS AND/OR ITS CONTRIBUTORS BE LIABLE TO ANY OTHER PARTY FOR LOST PROFITS OR ANY FORM OF DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THIS DELIVERABLE OR ITS GOVERNING AGREEMENT, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND WHETHER OR NOT AOUSD, ITS MEMBERS AND ITS CONTRIBUTORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2 Foreword

This document provides a formal specification of the core of the Universal Scene Description (USD) data representation, and scene construction algorithms.

This core specification describes the make-up of a scene description document, how to read its associated file formats and compose the scene description elements into a final stage. This document may be referred to as the “USD Core Specification” , or just “the Specification” or “the specification” hereafter.

This Specification is derived from the [OpenUSD](#) project developed by Pixar Animation Studios. References to OpenUSD in this Specification will specifically refer to the open-source project itself to disambiguate from USD as described in this Specification.

2.1 Domain Specific Specifications

This Specification is focused on the core aspects of USD. It does not describe domain specific uses of USD, such as:

- Geometry
- Materials
- Physics etc...

Domain specific uses of USD will be covered by other specifications published by the Alliance for OpenUSD (AOUSD).

2.2 Document Conventions

The document uses the following keywords as described in [BCP 14](#).

- **MUST**
- **MUST NOT**
- **REQUIRED**
- **SHALL**
- **SHALL NOT**
- **SHOULD**
- **SHOULD NOT**
- **RECOMMENDED**
- **MAY**
- **OPTIONAL.**

Any references to external documents are considered normative if the Specification uses any of the normative terms defined in this section to refer to them, or their requirements, either as a whole or in part.

Some language in the specification is purely informative, with an intention to give more clarity or suggestions for readers. If the entirety of a section is informative, it will be labelled as “Informative”, otherwise all sections must be considered normative.

All notes and examples are purely informative.

2.3 Supplemental Materials

In addition to this specification, we provide sample implementations. These will be made available at aousd.org.

2.4 Versioning

This specification is designed as version 1.0.1. Our document version is similar to [semantic versioning](#) and uses the MAJOR.MINOR.PATCH pattern.

The entirety of this specification will be versioned together as a whole, however the [file format sections](#) will also include their own version information that corresponds with the serialization version. The formats will list their own versioning semantics as well.

Changes that would change compatibility with previous versions of the specification will result in a MAJOR version changing.

The MINOR version will increase when the specification adds new information while preserving backwards compatibility with prior MINOR versions of the same MAJOR version.

PATCH versions will increase when there are corrections or changes to this specification that do not necessitate a MAJOR or MINOR change.

PATCH versions that increment due to corrections may break compatibility with prior PATCH versions of that MAJOR.MINOR release, if the spirit of the corrections are intended to preserve the MAJOR.MINOR versioning behaviour.

Please reference the [Changelog](#) to see notable changes between published versions. Changelogs will contain the changes relative to the prior release within the same MAJOR.MINOR.PATCH version family.

2.4.1 Referencing this Specification

Other documents that reference this specification must concretely specify the full version of this specification to help resolve any divergence there may be between PATCH versions.

It is optional for documents referencing this specification to adopt this specification's versioning semantics, but those documents should be aware of the compatibility differences as described above.

Documents that reference this specification must assume that they will work with future PATCH version changes of this specification within the specified MAJOR.MINOR version.

2.5 Errata and Support

Despite our best efforts, this specification may contain errata , or may require further clarifications.

We encourage readers to post on our forum at forum.aousd.org with any questions or concerns that they may have.

As noted in the [Versioning](#) section above, corrections issued for errata will lead to a PATCH increment on the specification version.

2.6 Changelog

Notable changes to the document version are listed here for reference.

2.6.1 1.0.1

- Initial release of the specification.

3 Introduction

Universal Scene Description (USD) is an authoring and interchange format for representing 3D scenes. Its layered composition model enables collaborative workflows and allows powerful ways to combine assets into larger assemblies. This specification describes the way that data is represented, authored, and composed into a final stage representing the 3D scene.

3.1 Representation and Authoring

Scene elements are represented by *scene description* that is organized into *scene description specifications*, hereafter referred to as *specs*. Specs are organized hierarchically, with the *layer spec* defining the root of the hierarchy and *prim*, *property*, *variant set*, and *variant* specs defining the elements in that hierarchy. Within a layer, specs are addressable via *paths*, which form the addressable representation of a spec from root (the layer spec) to leaf (the spec being addressed). Data is stored in specs via *metadata fields*, and *authoring* is done by providing values for those fields on the desired spec. The values provide *opinions* on the final value for a *scene object*, a composite object formed by *composing* all opinions on all specs at a given path when constructing the scene.

3.2 Scene Construction and Composition

A *stage* is formed by *composing* all specs reachable from a *root layer* into individual *scene objects*. The scene object representing the root layer forms the addressable root of the resulting stage. All specs reachable from the root layer are then composed to form the analogous scene object representation inside of the stage; e.g., *prims* and *properties*. The process of composition may result in opening additional layers to obtain further opinions. The scene object, which aggregates opinions on all specs addressable with the same path, has its opinions ordered from the spec providing the strongest opinion to that providing the weakest according to the strength ordering of its associated *composition arcs*. Each scene object in the final stage is also addressable via a path; that is, a path can be interpreted both on a stage (referring to aggregate scene objects) and an individual layer (referring to specs within that layer).

3.3 Specification Structure

This specification normatively describes:

- the data model and algorithms necessary to represent the data associated with specs
- the algorithms used to compose those specs into scene objects in a stage
- the means to extend the data model to contain new data
- the representation of the data model in formats for interchange

The specification begins with the **foundational data types** by describing the types that can be represented in USD and follows with the **document data model**, specifying the data fields

(and their types) that represent the data model for each spec. Building on that representation, the specification describes the methodology for addressing the contents of a layer through [paths](#) and [assets](#). The specification then describes the means for interpreting that data to [populate a stage](#) with scene objects that are created by [composing](#) the information in each spec contributing opinions to that scene object. The specification discusses the algorithms required to correctly resolve values from those scene objects through [value resolution](#). Finally, the specification describes how to extend the data model via [schemas](#) and how to store that data in [formats](#) suitable for interchange. At each step along the way, the reader is encouraged to use the [glossary](#) as a reference for the nomenclature introduced throughout.

4 Compliance

4.1 Interpreting Compliance

An implementation of the specification may be understood to be in compliance with the specification if the following are true:

- If the specification has a conformance test, and the implementation passes this conformance test.
- The result of checking compliance matches the written specification.
- If OpenUSD is checked for compliance on the same aspect, the same result is obtained as for the implementation.

4.2 Compliance Ambiguities

If the specification does not contain language describing a particular aspect, making interpretation necessary, the OpenUSD implementation should be considered as the expected behavior. Any such circumstance should be considered an unintentional omission in the specification, and may be addressed by the release of Addenda, Errata, or new releases of the specification.

4.3 Compliance Disagreements

- If a specification provided unit test checks for a result that disagrees with the specification, one or both of them are in error.
- If an implementation yields a result that differs from the specification, the implementation is in error.
- If an implementation yields a result that differs from the specification but matches the OpenUSD reference behavior, then one or more of the following is in error: the unit test, the specification, or OpenUSD itself.

Any such condition of disagreement may be addressed by the release of Addenda, Errata to the specification, new releases of the specification, or updates to the OpenUSD implementation, as deemed most appropriate.

4.4 Compliance Rubric

A compliant implementation of Universal Scene Description core specification must pass the following conformance tests:

1. Composed Stage Population
2. Value Resolution
3. Format Implementations (usda, usdc, usd, and usdz)

Conformance tests will be made available at aousd.org.

4.4.1 Composed Stage Population Compliance

Compliant implementations of composed stage population must populate a stage of composed scene graph objects (prims and properties) from the set of compliance root layers. For validation, a “squashed” version of the stage into a single layer will be provided. A “squashed” stage has removed all composition operators. A valid implementation of composed stage population produces the exact same set of a) scene graph objects and b) authored field values as the “squashed” layer.

Floating point fields must be exact at authored time samples, and exact at spline knots. Although interpolant methods such as linear interpolation, bezier evaluation and others are common and allowable, they are not subject to specification or compliance validation at the time of this writing.

Note: Squashing of composition operators may produce different results if the squashed layer is included via reference, payload, or subLayers into another stage because instancing and variant sets are dependent on presence of arcs.

4.4.2 Value Resolution Compliance

Compliant implementations of value resolution must sample values on scene graph objects into a “sampled” layer. A valid implementation of composed stage population produces the exact same set of a) scene graph objects and b) authored field values as the “sampled” layer.

Compliant implementations of metadata values must encode the fallback values from either a prim definition or the document data model explicitly into the sample layer. This addresses value resolution of relationship target paths.

Compliant implementations of attribute value resolution must negotiate between default, timeSamples, spline, and clips and sample the results at specified times. A “sampled” stage removes the spline field and replaces timeSamples with interpolated values and default with the authored default value or an attribute’s fallback value. For integral and string typed attributes, the value resolution must be exact.

Floating point fields must be exact at authored time samples, and exact at spline knots. Interpolated values between exact values may deviate a small amount from an evaluation of the reference interpolator if no other evaluation subject to the exact value requirement is perturbed.

Note: Resampling a stage may produce different results when consumed as the sampling frequency is being changed and extremes may be elided. Resampling as defined here is for validation.

4.4.3 Format Implementation Compliance

Compliant implementations of the usda, usdc, usd, and usdz formats must be provided. Formats implementing the document data model must produce the same set of specs and authored metadata fields and values as the compliance test.

5 Glossary

This glossary captures technical terms and definitions used in the USD core specification.

5.1 Core Terms

5.1.1 Asset

Resource or container of resources (layers, textures, volumes, etc.) providing the data for a scene.

5.1.2 Attribute

A type of Property meant for holding forms of numeric, string, and array valued data. Data types can be scalar (single valued) or dimensioned (arrays). Attributes can contain data that can vary over time.

5.1.3 Authored/Unauthored

An authored Property or Metadata field provides values as opinions for value resolution.

An unauthored field does not provide its own value but may contribute a fallback value for value resolution.

5.1.4 Composition

The process that assembles opinions for a scene object at a given Path from multiple Layers together using Composition Operators to determine the relative strength ordering of those opinions.

5.1.5 Composition Operators

Layer features that provide instructions on how to combine data from other sources. Those sources may exist in the same Layer, or may come from another Layer. The shorthand notation for the different types (and strength order) of composition is LIVERPS.

Operators which compose a subgraph into the local layer stack are more specifically termed “composition arcs”.

5.1.6 Core Specialized Type

A type that represents a higher-level data structure that cannot otherwise be handled by a **foundational data type**. For example, a Reference field is a Core Specialized Type that has an Asset Path and a Prim Path.

5.1.7 Document

A USD layer (see Layer).

5.1.8 Fallback Value

A value for a property or metadata field that may be used by consumers when that field has no value set (an “unauthored” field).

5.1.9 Flatten

The process of combining all Layers in a Stage into a single Layer, applying Composition and then removing [Composition Operators](#) so that a layer has no external dependencies. A flattened stage may preserve or introduce internal composition arcs to preserve behaviors such as instancing. See also [squash](#).

5.1.10 Format (Layer Format)

An implementation of the Layer document model. Formats are mappings from storage to a hierarchy of specs and their Metadata fields.

5.1.11 Inherits

A Composition Arc that aggregates scene description from a base Prim at a scene graph location. Unlike a Reference, Inherits arcs will aggregate opinions from the specified scene graph location through all levels of referencing.

5.1.12 Kind

Prim level metadata used to provide classification of Prims necessary for scene and asset management.

5.1.13 Layer

A fundamental container that contains zero or more hierarchical Prim elements of a Scene.

5.1.14 Layer Stack

The ordered set of Layers resulting from the recursive gathering of all Sublayers of a Layer, plus the Layer itself as first and strongest.

5.1.15 List Ops

List Operations, or ListOps, are value representations of operations (e.g. appended, deleted, prepended) that edit a list of items, producing orderings of unique elements of the same type.

5.1.16 LIVERPS

An acronym for the different Composition Operators in USD: Local, Inherits, VariantSets, rElocates, References, Payload, and Specializes. The acronym lists the strength ordering for how Composition composes Opinions and Namespaces.

5.1.17 Local

A Composition Operator that simply acquires Opinions from the local Layer Stack, including Sublayers.

5.1.18 Metadata

Lightweight name and typed value scene description data. Metadata cannot contain data that can vary over time.

5.1.19 Namespace

The set of Prim Paths that provide the identities for prims on a Stage, or Prim Specs in a Layer.

5.1.20 Opinions

Values specified in a Layer, commonly on Prim or Property specs.

5.1.21 Path

Identifies a location within a namespace (such as a prim on a stage). Paths in Universal Scene Description adhere to the [Path Grammar](#).

5.1.22 Payload

A Composition Arc that uses a target Path to aggregate scene data, similar to References, but allows easy control over whether to load the target scene data, enabling efficient handling of complex Scenes.

5.1.23 Prim

A container object represented by a unique Path within the Namespace hierarchy of a Stage that can contain a set of Properties. Prims also serve as the site of Composition arcs, and as such can define Composition arcs as part of its definition.

5.1.24 Prim Spec

Contributor of values (opinions) for a Prim in a Layer.

5.1.25 Property

A named data holder for a prim. Properties are the most granular data holder in a USD Layer. There are two types of Properties: Attributes and Relationships.

5.1.26 Property Spec

Contributor of values (opinions) for a Property in a Layer.

5.1.27 References

A Composition Arc that specifies a target Path and aggregates scene description from that path into the referencing Layer. References may point to a separate Layer, or to another Prim within the current Layer.

5.1.28 Relationship

A Property that creates a dependency between scene objects by allowing a Prim to target other Prims, Attributes, or Relationships.

5.1.29 Relocates

A Composition Arc that maps a Prim Path defined across a Composition Arc to a different Path in the local Namespace.

5.1.30 Scene

A set of USD data organized for a specific purpose. More specifically, a scene graph of Prims along with Metadata.

5.1.31 Schema

An object that defines and applies semantic meaning to populated scene elements via Properties, Metadata, and allowable behaviors. Schemas can act as templates to apply meaning and categorization to Prims and Properties.

5.1.32 Schema domain

A grouping of related Schemas.

5.1.33 Specializes

A Composition Arc that aggregates scene description from a base Prim at a scene graph location. Unlike a Reference, Specialize arcs will aggregate opinions from the specified scene graph location through all levels of referencing. Behaves similar to the Inherits Arc except for its relative position in the LIVERPS strength ordering.

5.1.34 Squash

The process of combining all Layers in a Stage into a single Layer, applying Composition to remove *all* [Composition Operators](#) so that a layer does not utilize composition or population features. See also [flatten](#).

5.1.35 Stage

An instance of the USD data model that holds the fully composed scene graph and associated data.

5.1.36 Stage Population

The process of creating a Stage by hierarchical creation of scene objects built from composed opinions of different specs for a prim at each path.

5.1.37 Sublayer

A Composition Operator which recursively introduces opinions from one Layer into a Layer Stack. The Layer Stack provides the “Local” opinions in the LIVERPS strength ordering.

5.1.38 Token

A handle for a registered string, that can be compared, assigned, and hashed in constant time.

5.1.39 Value Resolution

The act of taking the various Opinions at a Path and finding which Opinions are strongest, and therefore which contribute to the active value at that Path.

5.1.40 Variant

A named set of Opinions that represent an alternative representation or configuration for a Prim.

5.1.41 Variant Set

A named group of Variants.

5.1.42 VariantSets

A Composition Arc that uses a Variant Set and applies Opinions from a selected Variant from that set.

5.2 Externally Defined Terms and Definitions

The following terms and definitions are referred to, but not defined in the core specification. These terms originate from existing standards or established technologies. Where appropriate, links to existing standards for detailed definitions are provided.

5.2.1 AOUSD

The Alliance for OpenUSD ([AOUSD](#)) standards organization.

5.2.2 Axis

In computer graphics, a line used to position objects in 2D or 3D space. Examples: x-axis, y-axis, z-axis, up-axis.

5.2.3 Case Folding

The process of converting all characters in a collection of characters (string, etc.) into the same case, either all upper case or lower case.

5.2.4 Endianness

The order in which bytes within a word of digital data are addressed in memory.

5.2.5 Handedness

A vector space orientation. Examples: Right-handed, left-handed.

5.2.6 JSON

“JavaScript Object Notation”, a standard data interchange format, as described [here](#).

5.2.7 LZ4

A lossless compression algorithm defined [here](#).

5.2.8 OpenUSD

The [OpenUSD](#) project developed by Pixar Animation Studios.

5.2.9 PEG notation

“Parsing Expression Grammar” notation, a notation used for USD grammar, as defined in: Ford, Bryan (January 2004) “Parsing Expression Grammars: A Recognition Based Syntactic Foundation”. Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM. pp. 111-122

5.2.10 Quaternion

In computer graphics, a compact representation of the rotation of an object in three dimensions.

5.2.11 Scene graph

A collection of nodes, typically in a graph or tree structure, that logically represents a graphical scene.

5.2.12 Unicode

A text encoding standard as described in [The Unicode Standard, Version 15.1.0](#).

5.2.13 UTF-8

“Unicode Transformation Format – 8-bit”, a Unicode character encoding standard, as defined [here](#).

5.2.14 ZIP file format

A compressed file format described [here](#).

6 Foundational Data Types

6.1 Background

This section specifies common types used in Universal Scene Description’s foundational data models [1]. These types are used within the core file formats to encode information as serializations in files and other persistent or dynamic storage. Composite types, i.e., higher-level data structures, are described elsewhere, with their respective schemas [2]; this section declares the foundational types from which composite types may be constructed.

6.2 Scalar Types

Scalar types represent single values. In the table that follows, types such as string are indicated. A string is considered a scalar type as it is a sequence of characters, treated as a single unit. In this context, the term “scalar” emphasizes that the string is a single value, as opposed to a collection or composite type.

The integer types below are described as uN for an unsigned two’s complement integer of N bits, and iN for a signed two’s complement integer of N bits. Float types are described as fN where N is the number of bits. The distribution of sign, mantissa, and exponent within those bits is described in the noted IEEE standards.

Type	Fundamental Type	Notes
asset	utf8 encoded string, not containing C0 controls (U+0000..U+001F) nor C1 controls (U+0080..U+009F)	Asset identifiers are subject to variable substitution and resolution to locate resources at runtime
bool	u1	0 is false, 1 is true
double	f64	per IEEE 754-1985
float	f32	per IEEE 754
half	f16	per IEEE 754-2008
int	i32	
int64	i64	
string	utf8 encoded string	
timecode	f64	A unitless value
token	utf8 encoded string	Expected amortized constant time hash and equality comparison
uchar	u8	
uint	u32	
uint64	u64	

Note: Values of type token are expected to be commonly used strings optimized for comparison and hashing. They often function as an alternative to an enumerated type. Note that the specification for token and string values is the same, but runtimes may choose different runtime and storage representations based on the expected usage.

6.3 Dimensioned Types

Arrays are indicated with brackets containing the dimension of the array, for example, `f32[2]` means an array of two `f32` values as described in the [Scalar Types](#) section. `f32[3,3]` means a two dimensioned array, in row-major order, such that `f32[i, j]` indexes the element in the i th row and the j th column. If a `f32[4, 4]` matrix contained a transformation matrix where the upper 3x3 indicated a rotation with no scale, then the translation would be found in the fourth row; if the matrix were laid out linearly in a single dimension array, row following row, the translation would be found in the 13th, 14th, and 15th elements of that linear array [3].

Vectors are treated as row vectors when multiplied by a matrix. Vectors pre-multiply matrices to effect transformations. Accordingly, transforming a vector v by a series of transformation matrices may be written as $v_t = v \cdot S \cdot R \cdot T$

The quaternions are arranged as three imaginary coefficients, followed by a single real coefficient. They are assumed to be normalized as they are intended to be used as orientations, not general purpose complex numbers. Quaternions are treated as right-handed, irrespective of the handedness of the prim they are on. Similarly, matrix multiplication is treated as right-handed wherever handedness is important; and cross products may also be assumed to be right-handed whenever they are used to compute an orientation.

Type	Fundamental Type	Notes
<code>double2</code>	<code>f64[2]</code>	(x,y)
<code>double3</code>	<code>f64[3]</code>	(x,y,z)
<code>double4</code>	<code>f64[4]</code>	(x,y,z,w)
<code>float2</code>	<code>f32[2]</code>	(x,y)
<code>float3</code>	<code>f32[3]</code>	(x,y,z)
<code>float4</code>	<code>f32[4]</code>	(x,y,z,w)
<code>half2</code>	<code>f16[2]</code>	(x,y)
<code>half3</code>	<code>f16[3]</code>	(x,y,z)
<code>half4</code>	<code>f16[4]</code>	(x,y,z,w)
<code>int2</code>	<code>i32[2]</code>	(x,y)
<code>int3</code>	<code>i32[3]</code>	(x,y,z)
<code>int4</code>	<code>i32[4]</code>	(x,y,z,w)
<code>matrix2d</code>	<code>f64[2, 2]</code>	(x,y)
<code>matrix3d</code>	<code>f64[3, 3]</code>	(x,y,z)
<code>matrix4d</code>	<code>f64[4,4]</code>	(x,y,z,w)

Type	Fundamental Type	Notes
quatd	f64[4]	(i, j, k), r Unit length
quatf	f32[4]	(i, j, k), r Unit length
quath	f16[4]	(i, j, k), r Unit length

6.4 Algebraic Types

Type	Notes
opaque	An algebraic type that is not serializable and not representable by other fundamental types

6.5 Semantic Aliases For Types

Semantic aliases are not foundational types but aliases that assign a semantic role to a foundational type. Core value behaviors are only predicated on the underlying type, but higher level constructs should respect the semantics applied to values (such as using different transform semantics to values with normal, point, and vector roles).

Semantic Alias	Underlying Type	Role	Notes
color3d	double3	color	RGB
color3f	float3	color	RGB
color3h	half3	color	RGB
color4d	double4	color	RGBA
color4f	float4	color	RGBA
color4h	color4h	color	RGBA
normal3d	double3	normal	(x, y, z), Unit length (advisory)
normal3f	float3	normal	(x, y, z), Unit length (advisory)
normal3h	half3	normal	(x, y, z) Unit length (advisory)
point3d	double3	point	(x, y, z) Indicates position
point3f	float3	point	(x, y, z) Indicates position
point3h	half3	point	(x, y, z) Indicates position
vector3d	double3	vector	(x, y, z) Indicates direction and length

Semantic Alias	Underlying Type	Role	Notes
vector3f	float3	vector	(x, y, z) Indicates direction and length
vector3h	half3	vector	(x, y, z) Indicates direction and length
frame4d	matrix4d	frame	3-D Coordinate Transformation Matrix
texCoord2d	double2	texCoord	(x, y)
texCoord2f	float2	texCoord	(x, y)
texCoord2h	half2	texCoord	(x, y)
texCoord3d	double3	texCoord	(x, y, z)
texCoord3f	float3	texCoord	(x, y, z)
texCoord3h	half3	texCoord	(x, y, z)
group	opaque	group	A proxy for multiple values

6.5.1 Type and Alias Agreement

Comparisons between a semantic alias and an underlying type are specified as “agreement”. An alias agrees with its underlying type. `double3` and `color3d` are not equivalent but they do agree. Two equivalent underlying types are considered trivially agreeable.

Agreement is specified for an alias and an underlying type or two underlying types but not two semantic aliases.

6.6 Container Types

6.6.1 Arrays

Describe random access arrays of scalar and dimensioned types by appending brackets (`[]`) to the type name. For example, a `float` array is specified as `float[]`, a `double3` array is specified as `double3[]`, and a `string` array is specified as `string[]`. An array may be any non-negative size, including empty. Arrays of algebraic types are not permitted.

Explicit size qualifiers are not used to restrict or specify the size of an array. For example, `float[9]` may not be used to describe an array of 9 floats. Describe sizes and restrictions using additional values or fields.

For an array with size `n`, an index of `0` references the first element, `1` references the second element, and `n - 1` references the last element.

Index based access of arrays of dimensioned types should return the entire dimensioned value.

6.6.1.1 Semantic Aliases and Arrays

A semantic alias may be annotated with brackets ([]) to indicate an array of values, where each element conforms to the specified alias. For example, `color3f[]` denotes an array of values of type `float3`, with each element associated with the `color3f` semantic. Agreement of arrays with a semantic alias is defined by the agreement of the alias with an underlying element type. For example, `color3f[]` agrees with `float3[]`.

6.6.2 Dictionaries

A dictionary is an unordered map of strings to heterogenous values. Each key maps to one and only one value. Dictionaries must support the following as value types.

- Scalar types
- Dimensioned types
- Arrays of scalar or dimensioned types
- Dictionaries

They should not store semantic aliases for values. They should not store list operations (see below).

Dictionaries containing other dictionaries are considered “nested”.

An empty string is a valid key but not recommended.

dictionary is the normative name of this container type.

6.6.2.1 Combining ($\cup$)

Combining of two dictionaries in the domain of dictionaries $\mathbb{D}$ is a closed and associative operation.

A stronger dictionary S merged with a weaker dictionary W must contain

- Key value pairs from S where the key is not in W ’s keys
- Key value pairs from S where the key is in W ’s keys but both values aren’t dictionaries
- For every key in both S and W where both values are dictionaries, a new key value pair with the dictionary values combined
- Key value pairs from W where the key is not in S ’s keys

$$\begin{aligned}
 \cup : (\mathbb{D}, \mathbb{D}) &\rightarrow \mathbb{D} \\
 S &= \{(k_{s_0}, v_{s_0}), \dots, (k_{s_n}, v_{s_n})\} \\
 W &= \{(k_{w_0}, v_{w_0}), \dots, (k_{w_m}, v_{w_m})\} \\
 S \cup W &\equiv \{(k_{s_x}, v_{s_x}) \mid \forall (k_{s_x}, v_{s_x}) \mid k_{s_x} \notin \{k_{w_0}, \dots, k_{w_m}\}\} \cup \\
 &\quad \{(k_{s_y}, v_{s_y}) \mid \forall ((k_{s_y}, v_{s_y}), (k_{w_y}, v_{w_y})) \mid k_{s_y} = k_{w_y} \wedge \neg(v_{s_y}, v_{w_y} \in \mathbb{D})\} \cup \\
 &\quad \{(k_{s_z}, v_{s_z} \cup v_{w_z}) \mid \forall ((k_{s_z}, v_{s_z}), (k_{w_z}, v_{w_z})) \mid k_{s_z} = k_{w_z} \wedge v_{s_z}, v_{w_z} \in \mathbb{D}\} \cup \\
 &\quad \{(k_{w_q}, v_{w_q}) \mid \forall (k_{w_q}, v_{w_q}) \mid k_{w_q} \notin \{k_{s_0}, \dots, k_{s_n}\}\}
 \end{aligned}$$

6.6.3 List Operations

List operations or “list ops” are values that produce user orderings of unique elements of the same type.

Element uniqueness requires types that have robustly defined equality operations. For performance, implementations may require a well-defined hash function as well.

The following types (int, int64, uint, uint64, string, and token) are fundamental data types that must be supported by list operations. Floating point types should not be supported by list operation.

The set of types that support list operation for the specification may be extended to additional types (such as *scene object paths*) to support composition and other core features.

6.6.3.1 Shorthand

listop<\$TYPE> may be used as shorthand to refer to a list op that produces an ordering of elements of type \$TYPE. For example, listop<int> or listop<token>.

6.6.3.2 Definition

A single list operation L in the domain of list operations $\mathbb{L}$ is either “explicit” or “composable”.

$$\begin{aligned}\mathbb{L} &= \mathbb{L}_{\text{explicit}} \cup \mathbb{L}_{\text{composable}} \\ \mathbb{L}_{\text{explicit}} \cap \mathbb{L}_{\text{composable}} &= \emptyset\end{aligned}$$

List operations are compromised of subfields containing sequences of user ordered unique elements. Notationally, $\langle a, b, c, \dots \rangle$ should be read as a sequence where the user ordering of elements is important and $\{a, b, c\}$ should be read as a set where ordering does not matter.

An explicit list operation E holds a single explicit sequence of unique elements or items.

$$\begin{aligned}E \in \mathbb{L}_{\text{explicit}} &\equiv (\text{explicit} : \langle e_0, e_1, \dots, e_n \mid e_x \notin \{e_0, \dots, e_{x-1}\} \rangle) \\ E &= (\text{explicit} : \langle 1, 6, 5 \rangle)\end{aligned}$$

A composable list operation holds three unique operation sequences of appended, prepended, and deleted items.

$$\begin{aligned}C \in \mathbb{L}_{\text{composable}} &\equiv (\text{append} : \langle a_0, a_1, \dots, a_n \mid a_x \notin \{a_0, \dots, a_{x-1}\} \rangle, \\ &\quad \text{prepend} : \langle p_0, p_1, \dots, p_n \mid p_x \notin \{p_0, \dots, p_{x-1}\} \rangle, \\ &\quad \text{delete} : \langle d_0, d_1, \dots, d_n \mid d_x \notin \{d_0, \dots, d_{x-1}\} \rangle)\end{aligned}$$

Uniqueness is scoped to each individual sequence component of a list operation. The same element may appear in appended, prepended, and deleted item sequences for the same list operation.

$$C = (\text{append} : \langle 1, 2, 3 \rangle, \text{prepend} : \langle 1 \rangle, \text{delete} : \langle 3 \rangle)$$

6.6.3.3 Default Value

The normative default value (I') of a list op is a composable list operation with no elements in its appended, prepended, and deleted operation sequences.

$$I' \equiv (\text{append} : \langle \rangle, \text{prepend} : \langle \rangle, \text{delete} : \langle \rangle)$$

An explicit element sequence containing no elements is still considered valued and is a distinctly different state from the normative default. While under iteration, they both yield no elements, they produce different results when combining.

$$I' \neq (\text{explicit} : \langle \rangle)$$

All list ops without a valued explicit element sequence are considered “composable”.

$$L \in \mathbb{L}_{\text{explicit}} \Leftrightarrow L \notin \mathbb{L}_{\text{composable}}$$

6.6.3.4 Iteration

Iteration over a single explicit list operation E yields the explicit element sequence.

$$\text{iterate}(E) = \langle e_0, e_1, \dots, e_n \rangle$$

Iteration over a single composable list operation C yields all prepended elements $\langle p_0, p_1, \dots, p_n \rangle$ not in appended elements $\langle a_0, a_1, \dots, a_n \rangle$ followed by all appended elements.

$$\text{iterate}(C) = \langle p_0, p_1, \dots, p_n \mid p_x \notin C_{\text{append}} \rangle + \langle a_0, a_1, \dots, a_n \rangle$$

The deleted element sequence is always ignored for the purpose of iterating over the elements of a single list operation.

6.6.3.5 Equivalence (=)

Two list operations are equivalent if their ordered operation sequences are equivalent. Equivalence is denoted with the = symbol.

Two explicit list operations E and F are equivalent if their explicit operation sequences E_{explicit} and F_{explicit} are equivalent.

$$(E, F \in \mathbb{L}_{\text{explicit}}) \wedge (E_{\text{explicit}} = F_{\text{explicit}}) \Rightarrow E = F$$

Two composable list operations (C and D) are equivalent if their appended (C_{append} , D_{append}), prepended ($C_{prepend}$, $D_{prepend}$), and deleted (C_{delete} , D_{delete}) operation sequences are equivalent.

$$(C, D \in \mathbb{L}_{\text{composable}}) \wedge (C_{append} = D_{append}) \wedge (C_{prepend} = D_{prepend}) \wedge (C_{delete} = D_{delete}) \implies C = D$$

Equivalent list operations must be both explicit or both composable.

$$A \in \mathbb{L}_{\text{explicit}} \wedge B \in \mathbb{L}_{\text{composable}} \implies A \neq B$$

6.6.3.6 Combining ($\sqcup$)

List operators may be combined into one if they hold elements of the same type. The combining operation is denoted by the $\sqcup$ symbol.

Combining list operations A and B is closed and produces a new list operation C .

$$A \sqcup B \rightarrow C$$

Combining a stronger explicit list with any weaker list operation produces a new explicit list operation with the same explicit elements as the stronger list operation.

$$S \in \mathbb{L}_{\text{explicit}} \implies S \sqcup W \rightarrow S$$

Combining a stronger composable list operation S to an explicit list op E produces a new explicit list operation $S \sqcup E$.

$$(S \in \mathbb{L}_{\text{composable}}) \wedge (E \in \mathbb{L}_{\text{explicit}}) \implies (S \sqcup E) \in \mathbb{L}_{\text{explicit}}$$

$S \sqcup E$ must contain the following ordering of explicit items

- All $S_{prepend}$ items not in S_{append}
- Followed by all $E_{explicit}$ items not in S_{append} , S_{delete} , or $S_{prepend}$
- Followed by all S_{append} items

As an example, if $S = (delete : \langle 5 \rangle, prepend : \langle 5 \rangle, append : \langle 5 \rangle)$, and $E = (explicit : \langle 4, 5, 6 \rangle)$, the result of $S \sqcup E$ should be a new explicit list ($explicit = \langle 4, 6, 5 \rangle$) because append is the last operation applied.

Applying a composable list operation S to another composable list operation C produces a new composable list operation $S \sqcup C$.

$$S, C \in \mathbb{L}_{\text{composable}} \implies (S \sqcup C) \in \mathbb{L}_{\text{composable}}$$

$S \sqcup C$ must contain the following orderings of items

- $(S \sqcup C)$ has its deleted items ordered by
 - All C_{delete} items not in $S_{prepend}$ or S_{append}
 - Followed by S_{delete} items not in $S_{prepend}$, S_{append} , or C_{delete}
- $(S \sqcup C)$ has prepended items ordered by
 - All $S_{prepend}$ items not in S_{append}
 - Followed by all $C_{prepend}$ items not S_{append} , S_{delete} , or $S_{prepend}$
- $(S \sqcup C)$ has appended items ordered by
 - All C_{append} items not in S_{append} , S_{delete} , or $S_{prepend}$
 - Followed by all S_{append} items

6.6.3.7 Congruence ($\cong$)

Two list ops A and B are congruent if they yield the same iteration order under combination with any other list ops $L \in \mathbb{L}$. Congruence is denoted by the $\cong$ symbol.

$$A \cong B \equiv (\text{iterate}(A \sqcup C) = \text{iterate}(B \sqcup C)) \wedge (\text{iterate}(C \sqcup A) = \text{iterate}(C \sqcup B)) \forall C \in \mathbb{L}$$

Equivalent list operations are inherently congruent.

$$A = B \implies A \cong B$$

If two list ops are congruent and one is explicit, the other is necessarily explicit. If two list ops are congruent and one is composable, the other is necessarily composable.

$$(A \in \mathbb{L}_{\text{explicit}}) \wedge (A \cong B) \implies (B \in \mathbb{L}_{\text{explicit}})$$

$$(A \in \mathbb{L}_{\text{composable}}) \wedge (A \cong B) \implies (B \in \mathbb{L}_{\text{composable}})$$

If two list ops are congruent and one is explicit, they are equal.

$$(A \in \mathbb{L}_{\text{explicit}}) \wedge (A \cong B) \implies A = B$$

6.6.3.7.1 Congruent But Not Equivalent List Operations

$$\begin{aligned}
 C_1 &= (\text{append} : \langle 10, 50 \rangle, \quad \text{prepend} : \langle 10 \rangle, \quad \text{delete} : \langle 50 \rangle \quad) \\
 C_2 &= (\text{append} : \langle 10, 50 \rangle, \quad \text{prepend} : \langle 10 \rangle, \quad \text{delete} : \langle \rangle \quad) \\
 C_3 &= (\text{append} : \langle 10, 50 \rangle, \quad \text{prepend} : \langle \rangle, \quad \text{delete} : \langle \rangle \quad) \\
 C_4 &= (\text{append} : \langle 10, 50 \rangle, \quad \text{prepend} : \langle \rangle, \quad \text{delete} : \langle 50 \rangle \quad) \\
 C_5 &= (\text{append} : \langle 10, 50 \rangle, \quad \text{prepend} : \langle 10, 50 \rangle, \quad \text{delete} : \langle 50 \rangle \quad) \\
 C_6 &= (\text{append} : \langle 10, 50 \rangle, \quad \text{prepend} : \langle 10, 50 \rangle, \quad \text{delete} : \langle 10, 50 \rangle \quad)
 \end{aligned}$$

$$C_1 \cong C_2 \cong C_3 \cong C_4 \cong C_5 \cong C_6$$

The above elements are congruent because they all yield the same iteration order under combination with all other list ops. Specifically, authored elements in “prepend” and “deleted” are spurious if they also appear in “append”. Authored elements in “deleted” are spurious if they also appear in “prepend”.

$$\begin{aligned} C_m &= (\text{append} : \langle 10, 50 \rangle, \text{prepend} : \langle \rangle, \text{delete} : \langle \rangle) \\ C_n &= (\text{append} : \langle 50, 10 \rangle, \text{prepend} : \langle \rangle, \text{delete} : \langle \rangle) \\ C_m &\not\cong C_n \end{aligned}$$

The above elements are not congruent because their element ordering is different.

$$\begin{aligned} C_k &= (\text{append} : \langle 10 \rangle, \text{prepend} : \langle 50 \rangle, \text{delete} : \langle \rangle) \\ C_l &= (\text{append} : \langle 50, 10 \rangle, \text{prepend} : \langle \rangle, \text{delete} : \langle \rangle) \\ C_k &\not\cong C_l \end{aligned}$$

The above elements produce the same iteration order, but are not congruent because they’ll result in different ordering when combined with non-empty list ops.

6.6.3.7.2 Inert List Operations

An inert list op is a list operation which has no effect on traversal of elements.

Combining list operations with inert list operations may result in spurious prepended or deleted elements to be removed, yielding congruent but not always equivalent list operations.

Detecting inertness often requires context about the other operand. The default list op value was previously introduced as I' . I' is a trivially inert list operation.

Combining a stronger list operation S with the default list operation I' results in a list operation congruent with the stronger list operation.

$$S \sqcup I' \cong S$$

Combining the default list operation I' with a weaker list operation W results in a list operation congruent with the weaker list operation.

$$I' \sqcup W \cong W$$

Combining the default list operation I' with itself produces I' .

$$I' \sqcup I' = I'$$

6.6.3.8 Reducing

While individual element sequences must have unique elements, a single composable list operation may have elements appear in multiple operation sequences. An individual list operation is in its reduced form if no elements repeat across sequences.

$$\begin{aligned} \text{reduced}(C) \equiv & (\text{append} : \langle a_0, \dots, a_n \mid a_x \notin \{a_0, \dots, a_{x-1}\} \rangle, \\ & \text{prepend} : \langle p_0, \dots, p_m \mid p_x \notin \{p_0, \dots, p_{x-1}\} \cup \{a_0, \dots, a_n\} \rangle \\ & \text{delete} : \langle d_0, \dots, d_p \mid d_x \notin \{d_0, \dots, d_{x-1}\} \cup \{a_0, \dots, a_n\} \cup \{p_0, \dots, p_m\} \rangle) \end{aligned}$$

To reduce a composable list operation:

- Remove all elements in prepended from deleted
- Remove all elements in appended from prepended and deleted

Explicit list operations, as they only have one sequence, are inherently reduced.

Congruent reduced list operations are equivalent.

$$\begin{aligned} A \cong B &\implies \text{reduced}(A) \cong \text{reduced}(B) \\ A \cong B &\implies \text{reduced}(A) = \text{reduced}(B) \end{aligned}$$

6.6.3.9 Chaining ($\sqcup$)

An ordered chain of list operations A, B, C may be combined to produce a new list operation D .

$$\begin{aligned} A \sqcup B \sqcup C &\rightarrow D \\ \sqcup \langle A, B, C \rangle &\rightarrow D \end{aligned}$$

As the order of application matters, we can say that A is stronger than B and B is stronger than C .

Normative traversal of a chain of list ops should yield same ordering of elements as if the chain was combined to a single list operation.

Combining chains are associative but not commutative with respect to congruence.

$$\begin{aligned} (A \sqcup B) \sqcup C &\cong A \sqcup (B \sqcup C) \\ B \sqcup A \sqcup C &\not\cong A \sqcup B \sqcup C \end{aligned}$$

Since combining a stronger explicit list operation with any weaker operation produces a list

operation with identical explicit elements, combining a chain of list operations may terminate after the first explicit list operation in the sequence is encountered.

$$\bigsqcup \langle L_0, \dots, L_x, \dots, L_n \rangle = \bigsqcup \langle L_0, \dots, L_x \rangle \forall L_x \in \mathbb{L}_{\text{explicit}}$$

Any inert list operations may be dropped from a chain to produce a congruent (but not necessarily operation element equivalent) traversal. Note the default value I' is trivially inert and can always be dropped to produce a congruent traversal.

$$\begin{aligned} \bigsqcup \langle L_0, \dots, L_n \rangle &\cong \bigsqcup \langle L_0, \dots, L_n \setminus I' \rangle \\ \bigsqcup \langle A, I', B, C, I' \rangle &\cong \bigsqcup \langle A, B, C \rangle \end{aligned}$$

The normative definition of traversal prefers “congruence” over “equality” so implementers can choose different optimal implementations. When strict equality is required, note that congruence of reduced list operations implies equality.

$$A, B, C \in \mathbb{L}_{\text{reduced}} \Rightarrow \bigsqcup \langle A, I', B, C, I' \rangle = \bigsqcup \langle A, B, C \rangle$$

6.6.3.10 Deprecated Operations

There are two deprecated operations: reorder and add. These operations are not normatively specified. Supporting them would prevent list operations from being closed under combination. They may be discussed in other specifications for compatibility reasons.

Note: Hand authored example USD layers sometimes use a single deprecated add operation. This content should migrate to using append.

6.7 References

1. OpenUSD API “Sdf Metadata Types” https://openusd.org/release/api/sdf_page_front.html#sdf_metadata_types.
2. OpenUSD API “SdfSchema” <https://github.com/PixarAnimationStudios/OpenUSD/blob/v23.11/pxr/usd/sdf/schema.cpp#L427>.
3. OpenUSD API UsdGeom “Linear Algebra Basics” https://openusd.org/dev/api/usd_geom_page_front.html#UsdGeom_LinAlgBasics.

7 Document Data Model

7.1 Scope

This section specifies the document model of Universal Scene Description. Documents are resource backed descriptions of scenes termed a **layer**.

This section does not provide the specification for any of the required formats that will implement the layer document model. This does not generally provide a specification of composition, population, asset resolution, value resolution, or computation behavior but may note when fields or types are expected to be inputs to those higher level functions.

7.2 Layer Structure

Layers specify hierarchical elements of a scene called “specs” (scene description specifications). Specs contain values stored in named, typed “metadata fields”. Examples of specs include “prim specs” and “attribute specs”. Examples of metadata fields include documentation, references, or timeSamples.

All specs must be addressable by a unique absolute hierarchical identifier internal to the layer. This identifier is called a “path”. Paths have a textual representation defined by the **path grammar**.

Metadata fields are addressable in the context of a spec by a unique identifier. This identifier is the field’s “name”.

The contents of a layer consist of “values” addressable via a “spec path” and “field name” pair.

7.2.1 Example Layer Contents

The following snippet simulates the contents of a layer as a python or JSON dictionary but should not be taken as a literal implementation. Performant optimizations should take advantage of path element and field name redundancy as well as common access patterns and type validation. This snippet also elides required metadata fields.

```
={"/path/to/primSpec": {"someField" : 5, "anotherField" : "value"},  
  "/path/to/primSpec.propertySpec" : {"anotherField": "another value"}}
```

7.3 Specs

7.3.1 Forms

There are six concrete forms of hierarchical specs that may be specified in a layer:

- **Layer Spec**
- **Prim Specs**
- **Attribute Specs**
- **Relationship Specs**

- Variant Set Specs
- Variant Specs

Attribute specs and relationship specs share several metadata fields and are collectively termed “property specs”.

7.3.2 Hierarchy Semantics

Each spec in a layer is identifiable via a unique absolute path. The hierarchy of specs corresponds to the hierarchy of their paths.

All specs except layer specs have parent specs. All specs except property specs may have child specs. A parent spec idiomatically “contains” its child specs but this has no implication on layer storage.

Specs of the same form that share a parent are considered siblings. Attribute and relationship specs that share a parent may be considered sibling properties.

The specification promotes the efficient discovery of child specs and reserves **several metadata fields** for this purpose.

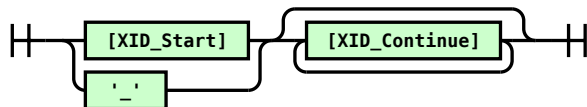
7.3.3 Names

Specs with parents (i.e., all specs except layer specs) have “names” that uniquely distinguish them from their siblings of the same form. Attributes and relationship specs with the same parent may not have the same name. It is otherwise valid for children of different forms to have the same name and parent.

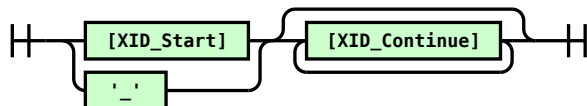
Spec names are UTF-8 encoded identifiers using the `XID_Start` and `XID_Continue` classes defined by the Unicode Standard. The following grammar defines the specific rules for each spec form.

```
Dot <- '.'
Minus <- '-'
ForwardSlash <- '/'
Colon <- ':'

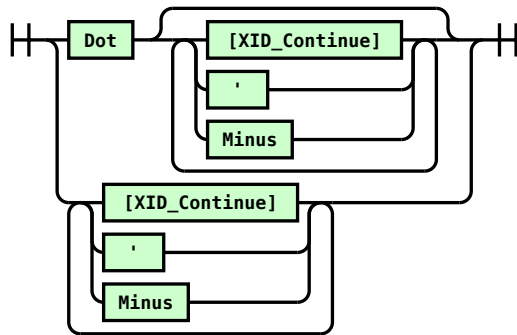
PrimName <- ([XID_Start] / '_' ) [XID_Continue]*
```



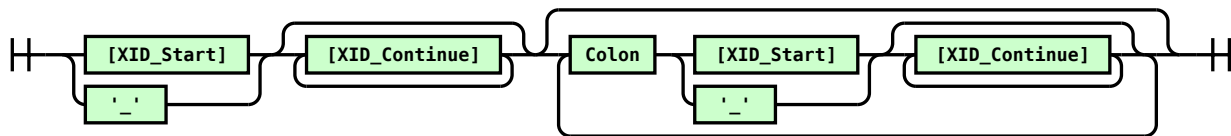
```
VariantSetName <- ([XID_Start] / '_' ) [XID_Continue]*
```



```
VariantName <- (Dot ([XID_Continue] / '|' / Minus)* ) / ([XID_Continue] / '|' / Minus)+
```



```
PropertyName <- ([XID_Start] / '_' ) [XID_Continue]* (Colon ([XID_Start] / '_' ) [XID_Continue]*)*
```



7.3.4 Layer Specs

Each layer contains one and only one layer spec. The single solidus character path (/ or U+002F) is reserved to identify the layer spec. A layer spec has no siblings, no parents, and no name.

A layer spec only contains prim specs as direct children. These are considered a layer's root prim specs. The layer spec is the immediate parent of all root prim specs.

As the ancestor of all root prim specs, the layer spec is sometimes also known as the "pseudo root spec" but the specification does not normatively use this term.

7.3.5 Prim Specs

Prim specs are organizing containers of the other specs in the hierarchy.

Prim specs may contain variant set specs, property specs, or other prim specs as direct children. Prim specs must not contain layer or variant specs.

7.3.6 Variant and Variant Set Specs

A variant set spec and its child variant specs create branches inside their parent prim spec with their values being conditionally considered by higher level composition and population functions.

Variant set specs can only contain variant specs. Variant specs may contain any spec a prim spec contains including other variant set specs.

7.3.7 Property Specs (Attribute and Relationship Specs)

Property specs are always leaf elements and must never have child specs. As leaves, they are the most granular holders of data in a layer.

There are two types of property specs: attribute specs and relationship specs.

Attribute specs primarily hold fundamental data types and arrays of those types whose values may vary over time. Relationship specs primarily store paths. The specifics of how these concepts are represented can be found in their more detailed definitions below.

7.4 Metadata Fields

A metadata field is defined as follows:

- Name
- Type
- Supported Spec Forms
- Fallback Value
- Authoring Requirement (Core Fields Only)
- Value Ranges (Core Fields Only)

Each spec form has a uniform set of allowable metadata fields. Each set of fields may be extended by implementations but should not vary between specs of the same form.

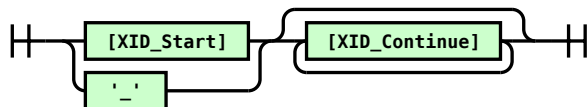
Each spec form (i.e., prim spec, relationship spec) may have a different set of supported metadata fields, but the type of named field must be globally consistent for all elements utilizing that field. As an example, not every spec form must support “documentation”, but it must be “string” for all spec forms that do.

7.4.1 Allowed Names

Metadata fields are UTF-8 encoded identifiers as defined by the Unicode Standard [2].

The standard provides the definition of the XID_Start and XID_Continue class of code points. Field names may start with a leading _ (U+005F).

```
FieldName <- ([XID_Start] / '_' ) [XID_Continue]*
```



7.4.2 Allowed Types

Metadata field storage must support the following types:

- Foundational Data Types
 - Any Foundational Scalar Type: float, int, ..., string, asset

- Any **Foundational Dimensioned Type**: float3, int3, ..., matrix4d, quath
- Any Foundational Scalar Type Array: float[], int[], ..., string[], asset[]
- Any Foundational Dimensioned Type Array: float3[], int3[], ..., matrix4d[], quath[]
- **dictionary**
- **listop** of the following types: int, int64, uint, uint64, token, string
- **Core Specialized Types**
 - **ObjectPath**
 - **Spline**
 - **References**
 - **Payloads**
 - **Retiming**
 - **Relocates**
 - **VariantValue**
 - **TimeSamples**
 - **EnumVariability**
 - **EnumSpecifier**

7.4.2.1 No Semantic Aliases

A metadata field must not use semantic aliases. For example, a metadata field can be a double3 but not a color3d, point3d, or vector3d. The layer document model requires that semantic aliases are stored in a **distinct typeName field** separately from the underlying value.

7.4.2.2 Core Specialized Types

Several core fields like references hold specialized types. Only the core fields described by the layer document model may use specialized types. Implementations may tailor support of core specialized types to these fields. Extension metadata fields described in other documents should not use core specialized types.

Specialized fields are used to describe types that have variant, enumerated option, compound data structure, or map semantics that are not otherwise handled by the foundational types. Definitions of specialized types may be found in the **core metadata fields** section.

Specialized types should not be supported in **dictionary** values. Specialized types may adopt list operation or array semantics (i.e. listop<SpecializedType> and SpecializedType[]). If a specialized type uses list operations, it must provide a well-defined equality operation.

Names used as shorthand to describe core specialized types should not be read as normative.

7.4.2.3 Speculative ObjectPath as a Core Specialized Type

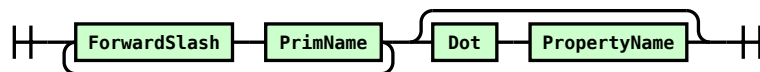
Several core metadata fields (and subfields) hold ObjectPath valued types. ObjectPath values speculatively target elements in the namespace of a layer. A speculative target is a target that might exist, but there's no requirement that it is an authored spec. Higher level

composition and population algorithms may impose additional validation restrictions on speculative targets, but these should not affect how a layer is read or written.

ObjectPath valued fields should only target prim or property specs. They should not target layer specs, variant set specs, or variant specs.

Every ObjectPath value has an injective mapping onto an absolute textual representation of the path as specified by the [path grammar](#). Note that the full path grammar is not needed for the purposes of the normative textual representation as ObjectPaths should not target layer, variant, and variant specs.

```
ObjectPath <- (ForwardSlash (PrimName))+ (Dot PropertyName)?
```



listop<ObjectPath> valued fields should be considered a specialized type as well. Path equivalence is defined having equivalent normative textual representations. Implementations may optimize this and are not required to store the full textual representation.

7.4.2.4 Cubic Splines Specialized Type

Cubic splines (hereafter referred to as simply “splines”) are data structures that represent a curved mapping function from time to value. Each spline is defined by a set of knots; points through which the curve passes with the shape of the curve controlled by tangents specified at the knots.

7.4.2.4.1 EnumCurveType

Value	Description
bezier	Defines the spline as a bezier curve
hermite	Defines the spline as a hermite curve

A spline can be either bezier or hermite.

7.4.2.4.2 EnumInterpolationMode

Value	Description
none	No interpolation value at this segment
held	Constant interpolation value at this segment
linear	Linear interpolation value at this segment
curve	Bezier or Hermite interpolation value at this segment, depending on spline curve type

Values along the curve are interpolated according to a chosen interpolation mode, which may vary along the curve between the knots. Interpolation can be held (constant value along the segment), linear (linearly interpolated value along the segment), curve (bezier or hermite interpolation along the segment), or none (implying a value block where no value exists for the segment).

7.4.2.4.3 SplineKnot

Subfield	Type	Default
time	double	0.0
preTangentSlope	double	0.0
preTangentWidth	double	0.0
postTangentSlope	T	0.0
postTangentWidth	T	0.0
nextInterpolationMode	EnumInterpolationMode	held
value	T	0.0
preValue	T	0.0

Each knot is associated with a time, a value at that time, and a pre and post value defining the slope of the tangent to the curve before and after the knot. A knot can be dual-valued, meaning there is a value discontinuity at the knot represented by an additional pre-value. If bezier, knots also store the time width of the pre and post tangent. The value, pre value, and tangent slopes may be represented by half, float or double and must be uniform in representation. Each knot stores an interpolation mode, which defines how the value is interpolated along the segment following the knot.

Implementations may optionally choose to store a dictionary of custom data with each knot.

7.4.2.4.4 EnumExtrapolationMode

Value	Description
none	No extrapolation value in this region
held	Constant extrapolation value in this region
linear	Linear interpolation based on edge knots
sloped	Linear interpolation with specified slope
looprepeat	Knot curve repeated, offset so ends meet
loopreset	Curve repeated exactly, discontinuous join
looposcillate	Like loopreset but every other copy reversed

An extrapolation mode defines how to retrieve a value outside the spline regions (i.e., before all knots and after all knots, referred to as pre-extrapolation and post-extrapolation). This mode may be held (constant value), linear (linear interpolation based on edge knots), sloped (linear interpolation with a specified slope), loop repeat (knot curve

is repeated, offset so the ends meet), loop reset (knot curve is repeated exactly, with discontinuous joins), loop oscillate (knot curve is repeated exactly, with discontinuous joins, but every other copy reversed), or none (implying a value block with no value).

7.4.2.4.5 LoopParameters

Subfield	Type	Default
protoStart	double	0.0
protoEnd	double	0.0
numPreLoops	int	0
numPostLoops	int	0
valueOffset	double	0.0

A spline may contain a maximum of one inner loop region. This defines the parameters of an inner loop.

7.4.2.4.6 Spline (Specialized Type)

Subfield	Type	Default
curveType	EnumCurveType	bezier
preExtrapolationMode	EnumExtrapolationMode	held
preExtrapolationSlope	double	0.0
postExtrapolationMode	EnumExtrapolationMode	held
postExtrapolationSlope	double	0.0
loopParameters	LoopParameters defaultsOf(LoopParameters)	
knots	SplineKnot[]	[]

Together, the above information defines a spline.

Implementations are free to optimize these data structures as they see fit.

7.4.3 Authored State

Metadata fields may be authored or unauthored. An authored metadata field holds a value that may be accessed by consumers of the layer. An unauthored field does not hold a value. Metadata fields should hold values that match the metadata field's declared type.

7.4.3.1 Required Fields

Specs are puzzle pieces that are intended to fit together to form a complete view of a scene. Most fields are considered electively authored in the context of an individual layer. Each spec form has a set of required fields that must be considered always authored.

A required field for the context of the document should be read as “required to consume this layer”. Electively specified fields are elective only for the context of consuming the in-

dividual layer. Higher level constructs may impose additional validation requirements, including requiring fields to have authored values.

As an implementation detail, any format may choose to elide explicit storage of any field (including required fields), but it still must present those fields as being authored if it intends to contribute an opinion.

7.4.4 Type Registry

Fields of the same name are required to have the same type. Implementations generally maintain a registry of field names and types.

Handling a stored value whose type differs from the field's registered type is layer format defined. Format implementations may derive an authored field's type from the registry when they share an internal representation, but transparent casting is otherwise discouraged.

For example, token and string types may share the same storage representations in both text and binary formats. Numbers may similarly have a shared storage representation with respect to precision.

7.4.5 Value Ranges

Implementations of the layer document model should generally not validate the value of a field beyond meeting the specification of its type. For example, implementations should not check if an asset path valued field exists when consuming the layer. Implementations should check that numeric value is in range when the range is a characteristic of the type.

Core metadata fields specified in the layer document model may impose additional per field restrictions on top of the type that implementations are expected to maintain for the purposes of reading and writing layers, but no other fields should impose restrictions.

Implementations may consider value restrictions in how they store fields. For example, a token typed field that is constrained to not have whitespace may elide quotes in its textual representation.

Additional validation restrictions may be imposed by higher level composition, population, and schema domain functions, but these should be not reflected in the reading and writing of layers.

7.4.6 Speculative Values

Many fields can be authored speculatively with semantics requiring a composed scene graph. It's generally valid for fields that speculatively store the names of children or paths that do not exist on the current layer. Fields that store names, scene paths, and asset paths should be presumed to be speculative unless otherwise noted.

7.4.7 Extension Metadata Fields

The layer document model allows the set of metadata fields to be implementation or schema domain extensible.

- The type of an extension field must never be a **specialized type**.
- Extension fields must not be **required** to consume a layer's contents.
- Extension fields must not impose additional **restrictions on allowed values** for the purposes of consuming a layer.

User and facility extension fields are generally discouraged in favor of leveraging existing user data dictionaries or user property namespaces which are more generally interchangeable.

Authoring and reading of an unknown extension metadata fields is implementation defined.

7.4.8 Fallback Values

Each named metadata field should specify its own fallback value that may be used by consumers when a field is unauthored.

Fallback value acquisition should not be transparent. Consumers must be able to distinguish between unauthored and authored for any non-required field.

Required metadata fields by definition cannot be in an unauthored state, but may specify a fallback to signify a preferred initial value.

Consumers may derive behavior based on authored state before deciding to acquire a field's fallback value. Specifying behavior based on the authored state is necessary for composition and stage population to function.

Fallback values are specified per metadata field identifier and may not vary across specs.

7.5 Formats

An implementation of the layer document model is called a "format". Formats are mappings from storage to a hierarchy of specs and their metadata fields.

Formats often correspond to specific file formats but are not required to be file backed. The core specification defines several required and elective formats for implementations. Implementations may allow additional formats to be defined as extensions.

Formats must be able to provide:

- A list of a layer's authored specs
- Values for all required metadata fields if valid
- Authored state for elective fields
- Values for elective fields if authored and valid

While every field has a reserved name that implementations may use for lookup, implementations may provide for specialized accessors or storage semantics that can optimize reads.

Performant formats should provide for partial reads of map and dictionary fields (including time samples).

Formats may provide an interface for reading a stored field whose name is not in a form's allowed set for the purposes of inspection, debugging, and repair.

7.5.1 Format Extensions

Unambiguous and performant format detection should be determined by extracting an extension from a **resolved asset location** for a layer.

Format specifications must declare and reserve one or more extensions for dispatching layers for read and write. Resource protocol specifications must specify how to extract an extension from a resolved asset location.

Format extensions should match the regular expression pattern `[a-z][a-z0-9]*`. Format extensions `usd[a-z0-9]*` should be considered reserved for detection and dispatch of formats defined by the core specification.

Extensions should be case-folded for the purposes of dispatch to a format's implementation. For example, assets with extensions USD, Usd, and UsD should use the `usd` format implementation.

7.5.2 Format Mapping

A format does not have to support all concepts described in the layer document model. Supported formats may be classified in terms of mappability of data between the layer document model and the format.

7.5.2.1 Partially Mapped Formats

Formats whose contents cannot be completely mapped into layer specs and fields are classified as partially mapped formats. Flattening and other transformations may not be able to regenerate the original data source. These are typically read only formats, but the specification doesn't forbid implementations from writing to them.

7.5.2.2 Fully Mapped Formats

A fully mapped format's contents can be completely mapped into layer specs. The test for a fully mapped format is being able to convert data to layer document model and back to the original format with equivalent content.

However, there may be concepts in the layer document model (like variant specs) that are not representable, so there may be loss when converting other formats to a fully mapped format.

7.5.2.3 Bijective Formats

A format is considered bijective if

- It can read and write all fields (including extension fields) for all spec forms (including variant specs)
- All of its contents can be read and written as fields

Bijjective formats are generally designed around the layer document model.

Authored values must be readable and re-serializable without any loss when converting between one bijjective layer format and another bijjective layer format. For formats which support multiple internal representations of data (such as multiple compression schemes), the internal representation is not required to persist across transformations.

7.5.2.4 Specialization: Forwarding Formats

A format may delegate its implementation to other formats and still meet the requirements of mapped and bijjective formats. A simple forwarding format delegates its implementation to one or more other formats. Forwarding formats can be bijjective when their delegates are also bijjective.

usd is an example of a forwarding format that forwards to either usda or usdc.

7.5.2.5 Specialization: Package Formats

An archive may present itself as a single layer. The layer may specify asset valued fields internal to the archive like images, and audio files. This includes specifying asset valued fields holding layers to be used in composition.

They are called out specially here, as a case of a format that is otherwise bijjective but whose contents may require special consideration under flattening and other layer transformations.

7.6 Core Metadata Fields

The following are considered core fields [1] with their identifiers reserved for usage across multiple domains including composition and stage population. Formats may tailor their support for core fields around allowed value ranges specified for each field but should otherwise support any valid value for a field's specified type.

7.6.1 Layer Spec Fields

7.6.1.1 Hierarchy Fields

These fields list the immediate children of the spec in the layer. These fields are non-speculative and should present the names of all child specs.

7.6.1.1.1 primChildren: token[]

Array containing the persistent ordering of names of all root prim specs.

Fallback value: []

Value Ranges:

- Format storage should keep this field in agreement with spec storage. Handling child prim specs stored but not specified in `primChildren` or values in `primChildren` without a corresponding prim spec is format defined. This is so formats have the flexibility on how to keep them in sync or defer their reading.

7.6.1.2 Composition Fields

The values of these fields are considered inputs to composition.

7.6.1.2.1 **subLayers: asset[]**

An array of weaker layers for composition to compose via LIVERPS.

Fallback value: []

Value Ranges:

- Elements of this array should non-empty

7.6.1.2.2 **subLayerOffsets: Retiming[] (Specialized Type)**

An array of per layer re-timings applied to time samples and any timecode valued fields.

Subfield	Type	Default
offset	double	0.0
scale	double	1.0

Fallback value: []

Value Ranges:

- This field should have the same length as `subLayers`. It's format defined what to do if they do not, allowing formats to read them either as independent fields or align their storage.

7.6.1.2.3 **defaultPrim: token**

The root prim to be used when a layer is targeted by composition arcs without an explicit prim identifier.

This field is speculative as the target may be introduced through composition.

A non-empty authored `defaultPrim` should be convertible to an absolute prim path.

- If `defaultPrim` is a valid prim path as specified by the path grammar:
 - If it starts with `/`, it is already an absolute path.
 - Otherwise, it's a path relative to `/`

Convertibility to an absolute prim path is not a value restriction applied by the layer document model.

Fallback value: ""

7.6.1.2.4 **layerRelocates: Relocates[] (Specialized Type)**

Layer relocates are used by composition to move elements in the scene graph to other path locations.

Subfield	Type	Default
source	ObjectPath	N/A
target	ObjectPath or unset	N/A

The fields are speculative as the relocate sources may be introduced through composition.

An empty target subfield implies the source is to be removed from the scene graph.

Fallback value: []

Value Ranges:

- The source path should target a prim path
- The target path, if authored, should target a prim path

7.6.1.3 **Population Fields**

Fields that affect how specs are populated or traversed.

7.6.1.3.1 **primOrder: token[]**

Preferred ordering of composed root prims identified by name. A sparse list implies the explicitly ordered elements should be traversed before absent ones. This should not affect the ordering of serialized specs.

This field's primary client is the composed scene graph and may speculatively include the names of root prims that are not specified in the reordering layer. The tokens in this field should only contain valid names for specs.

Fallback value: []

Value Ranges:

- Elements should be valid prim names.

7.6.1.4 **Timing Fields**

Fields related to how time samples on this layer should be interpreted. Some timing fields are advisory and others may affect value resolution.

7.6.1.4.1 **timeCodesPerSecond: double**

Number of discrete time codes that comprise a second. This may affect value resolution.

Note that while timeCodesPerSecond should be positive, it is not a formal restriction of the data model.

Fallback value: 24.0

7.6.1.4.2 framesPerSecond: double

Expected sampling frequency primarily for playback.

Fallback value: 24.0

Value Ranges:

- Formats should always provide a positive framesPerSecond value.

7.6.1.4.3 startTimeCode: double

Start of region of interest of time samples for this layer.

Fallback value: 0.0

7.6.1.4.4 endTimeCode: double

End of region of interest of time samples for this layer.

Fallback value: 0.0

7.6.1.5 User Interface Fields

7.6.1.5.1 documentation: string

User facing documentation about a layer that may be displayed in tooltips and other interfaces.

Fallback value: ""

7.6.1.6 User Fields

Users may store additional data in these fields.

7.6.1.6.1 comment: string

Container for user notes about a layer.

Fallback value: ""

7.6.1.6.2 customLayerData: dictionary

Additional user data about the layer. Generally preferable to store user or site data here than extend the set of metadata.

Fallback value: {}

7.6.1.7 Out of Scope Fields

These fields are reserved for inclusion in future specifications

- expressionVariables
- colorManagementSystem
- colorConfiguration

7.6.1.8 Deprecated Fields

These fields are reserved though their usage is deprecated.

- framePrecision
- hasOwnedSublayers
- owner
- sessionOwner
- startFrame
- endFrame

7.6.2 Prim Spec

7.6.2.1 Required Fields

These fields must always be specified for all instances of a spec.

7.6.2.1.1 specifier: EnumSpecifier (Specialized Type)

All prim specs must have a specifier field. Higher level composition, population, and traversal functionality take this into account though it does not have any semantic meaning at the layer level.

Value	Description
def	Concrete defining specifier
over	Sparse override specifier
class	Abstract defining specifier

While required, these fields have no semantic meaning at the layer levels. Composition and stage population specifications should describe how this type composes and impacts traversal of a populated stage.

Fallback value: over

7.6.2.2 Hierarchy Fields

These fields list the immediate children of the spec in the layer. These fields are non-speculative and must present the names of all child specs.

7.6.2.2.1 primChildren: token[]

Array containing the persistent ordering of names of all child prim specs.

Fallback value: []

Value Ranges:

- Format storage should keep this field in agreement with spec storage. Handling child prim specs stored but not specified in primChildren or values in primChildren without

a corresponding prim spec is format defined. This is so formats have the flexibility on how to keep them in sync or defer their reading.

7.6.2.2.2 **propertyChildren: token[]**

Array containing the persistent ordering of names for all child property specs.

Fallback Value: []

Value Ranges:

- Format storage should keep this field in agreement with spec storage. Handling child property specs stored but not specified in propertyChildren or values in propertyChildren without a corresponding property spec is format defined. This is so formats have the flexibility on how to keep them in sync or defer their reading.

7.6.2.2.3 **variantSetChildren: token[]**

Array containing the persistent ordering of names of all child variant set specs.

Fallback Value: []

Value Ranges:

- Format storage should keep this field in agreement with spec storage. Handling child variant set specs stored but not specified in variantSetChildren or values in variantSetChildren without a corresponding variant set spec is format defined. This is so formats have the flexibility on how to keep them in sync or defer their reading.

7.6.2.3 **Composition Fields**

These values are considered inputs to composition.

7.6.2.3.1 **references: listop<Reference> (Specialized Type)**

List operation contributing target inputs to LIVERPS composition.

Subfield	Type	Default
target	asset, ObjectPath, or (asset, ObjectPath)	N/A
offset	double	0.0
scale	double	1.0

A reference can target either an asset, an ObjectPath, or an (asset, ObjectPath) pair.

Equality of references for the purposes of resolving chains of list operations requires all fields be equal, including the floating point offset and scale fields. Approximate equality should not be used on offset and scale.

Implementations may support a fifth vestigial dictionary field for storing custom data but normatively specified behavior should not depend on this.

Implementations may share offset/scale data structures with the Retiming data. Targets could be implemented or stored as a variant. It may also be implemented as two optional subfields but is only valid when at least one of the subfields is set.

Fallback value: []

Value Ranges:

- ObjectPath targets should always be prim paths and never property paths.

7.6.2.3.2 payload: listop<Payload> (Specialized Type)

List operation contributing target inputs to LIVERPS composition. Despite the singular field name, the field is a list operation. An empty asset value implies the path is internal. An empty path implies that the target is defined by the asset's metadata. Either asset or path may be empty but both cannot be.

Subfield	Type	Default
target	asset, ObjectPath, or (asset, ObjectPath)	N/A
offset	double	0.0
scale	double	1.0

A payload can target either an asset, an ObjectPath, or an (asset, ObjectPath) pair.

Equality of payloads for the purposes of resolving chains of list operations requires all fields be equal, including the floating point offset and scale fields. Approximate equality should not be used on offset and scale.

Implementations may share offset/scale data structures with the Retiming data. The target subfield could be implemented or stored as a variant. It may also be implemented as two optional subfields but is only valid when at least one of the subfields is set.

Fallback value: []

Value Ranges:

- ObjectPath targets should always be prim paths and never property paths.

7.6.2.3.3 inheritPaths: listop<ObjectPath>

List operation contributing prim targets that serve as input to LIVERPS composition.

Paths may be speculative and target prims introduced through composition.

Fallback value: []

Value Ranges:

- ObjectPath targets should always be prim paths and never property paths.

7.6.2.3.4 **specializes: listop<ObjectPath>**

List operation contributing prim targets that serve as input to LIVERPS composition.

Paths may be speculative and target prims introduced through composition.

Fallback value: []

Value Ranges:

- ObjectPath targets should always be prim paths and never property paths.

7.6.2.3.5 **variantSetNames: listop<string>**

Names of child variant sets that may contribute to LIVERPS composition. The order of elements in the list op contributes to the strength order in LIVERPS.

Variants may be speculative and introduced through composition. Unlike variantSetChildren, this field is not required to be in agreement with the underlying spec storage.

Fallback value: []

Value Ranges:

- Elements should be valid variant set names.

7.6.2.3.6 **variantSelection: VariantSetMap (Specialized Type)**

Variant selection opinions for LIVERPS composition.

- Key Type: string
- Value Type: string

The key refers to the variant set name and the value refers to the variant selection of the map.

Selections may be speculative with target variants sets and variants introduced through composition.

An empty string is valid and implies that no variants should be considered by composition.

Fallback value: {}

Value Ranges:

- Keys should be valid variant set names.
- Values should be variant names or the empty string.

7.6.2.4 **Population Fields**

7.6.2.4.1 **typeName: token**

Identifier of a schema. Schemas are used to apply semantic meaning and allowable behaviors to populated scene elements.

typeName only provides an opinion about the schema type to be considered by composition and has no semantic meaning at the layer level.

Fallback Value: ""

Value Ranges:

- This field should be an ASCII identifier as specified by the regular expression `^[A-Za-z_][A-Za-z0-9_]*$`. Handling invalid type names that do not match this pattern is format defined.

7.6.2.4.2 active: bool

If false, this location should be considered removed from the scene graph allowing some composition and population behaviors to be skipped.

This has no semantic meaning at the layer level, and this field must not affect a layer's available specs.

Fallback value: true

7.6.2.4.3 instanceable: bool

If true, composition and population behaviors at this location may be shared and ignore descendant opinions.

This has no semantic meaning at the layer level, and this field must not affect a layer's available specs.

Fallback value: false

7.6.2.4.4 kind: token

Describes points of interest in the scenes as part of higher level constructs such as model hierarchy population.

This has no semantic meaning at the layer level.

Fallback value: ""

7.6.2.4.5 primOrder: token[]

Preferred iteration order of composed child prims identified by name. A sparse list implies the explicitly ordered elements should be traversed before absent ones. This should not affect the ordering of serialized specs.

This field's primary client is the composed scene graph and may speculatively include the names of prims that are not specified in the reordering layer. The tokens in this field should only contain valid names for specs.

Fallback Value: []

Value Ranges:

- Elements should be valid prim names.

7.6.2.4.6 **propertyOrder: token[]**

Preferred iteration order of composed child properties identified by name. A sparse list implies the explicitly ordered elements should be traversed before absent ones. This should not affect the ordering of serialized specs.

This field's primary client is the composed scene graph and may speculatively include the names of properties that are not specified in the reordering layer. The tokens in this field should only contain valid names for specs.

Fallback Value: []

Value Ranges:

- Elements should be valid property names.

7.6.2.5 **User Interface Fields**

Hints as to how specs should present in user interfaces. These properties should not affect composition, population, or imaging behaviors.

7.6.2.5.1 **displayName: string**

Hint as to how the prim may be presented to the user. This is any UTF-8 string and not subject to any of the identifier rules.

Fallback Value: ""

7.6.2.5.2 **displayGroupOrder: string[]**

Hint as to how display groups of properties should be ordered.

This field is speculative and may refer to display groups that are introduced through composition.

Fallback Value: []

7.6.2.5.3 **hidden: bool**

Hint of whether this prim should be presented to the user in user interface elements like scene graph tree views.

Fallback value: false

7.6.2.5.4 **documentation: string**

User facing documentation about a prim.

Fallback Value: ""

7.6.2.6 **User Fields**

Fields for storing user or site specific data.

7.6.2.6.1 customData: dictionary

Additional user data about the prim. Generally preferable to store user or site data here than extend the set of metadata.

Fallback Value: {}

7.6.2.6.2 assetInfo: dictionary

Additional user data that provides introspection about an asset's structure, such as an asset name, identifier, or version number. This field is advisory and must not participate in composition or asset resolution.

7.6.2.6.3 comment: string

Container for user notes about a prim.

Fallback Value: ""

7.6.2.7 Deprecated Fields

These fields are reserved though their usage is deprecated.

- relocates
- symmetryFunction
- symmetryArguments
- symmetricPeer
- suffix
- suffixSubstitutions
- prefix
- prefixSubstitutions
- permission

7.6.3 Property Spec

7.6.3.1 Required Fields

These fields must always be specified for all instances of a spec.

7.6.3.1.1 custom: bool

Declare whether a property was intended to be schema or user specified.

It is preferred that user defined fields are organized under the userProperties namespace than to rely on the custom field.

Fallback value: false

7.6.3.2 User Interface Fields

Hints as to how specs should present in user interfaces.

7.6.3.2.1 displayName: string

Hint as to how the property may be presented to the user. This is a UTF-8 string and not subject to any of the identifier rules.

Fallback Value: ""

7.6.3.2.2 displayGroup: string

Hint as to how this property may be organized when presenting to the user.

Fallback Value: ""

7.6.3.2.3 hidden: bool

Hint of whether this property should be presented to the user.

Fallback value: false

7.6.3.2.4 documentation: string

User facing documentation about how property, often defined in schemas.

Fallback Value: ""

7.6.3.3 User Fields

Fields for storing user or site specific data.

7.6.3.3.1 customData: dictionary

Additional user data about the property. Generally preferable to store user or site data here than extend the set of metadata.

Fallback Value: {}

7.6.3.3.2 assetInfo: dictionary

Data to provide introspection about asset structure primarily for asset valued properties. This field is advisory and must not participate in composition or asset resolution.

Fallback Value: {}

7.6.3.3.3 comment: string

Container for users notes about a property

Fallback Value: ""

7.6.3.4 Deprecated Fields

These fields are reserved though their usage is deprecated.

- symmetryFunction
- symmetryArguments

- symmetricPeer
- suffix
- prefix
- permission

7.6.4 Attribute Spec

Attribute specs inherit all fields from property spec.

7.6.4.1 Required Fields

These fields must always be specified for all instances of a spec.

7.6.4.1.1 typeName: token

Specifies the expected type of any authored default and time sample values. This may be a fundamental scalar, dimensioned, or algebraic type, as well as any associated semantic aliases. Arrays of scalar and dimensioned types (and their associated semantic aliases) are also allowed. (ie. typeName may be color3f[]).

Types that are allowable as metadata fields but not as attribute types include dictionaries and list ops.

Fallback Value: ""

Value Ranges:

- This field should be a foundational data type or semantic alias identifier constrained by the regular expression `^[A-Za-z_][A-Za-z0-9_]*(\[\\])?$. Handling invalid type names that do not match this pattern is format defined.`

7.6.4.1.2 variability: EnumVariability (Specialized Type)

Value	Description
varying	Resolved value of the attribute may but is not required to vary over time
uniform	Resolved value of the attribute is not expected to vary over time

This field is speculative and does not imply that timeSamples, spline, or default have been authored in the current layer (or ever).

Fallback Value: varying

7.6.4.2 Value Fields

These fields are considered the property's primary value to be resolved. Note it is not required for default, spline, or timeSamples to be authored. An attribute may be "specified" but not "valued".

7.6.4.2.1 default: VariantValue (Specialized Type)

Value that does not vary over time.

A variant value may store a value of one of the following types:

- Any **Foundational Scalar Type**: float, int, ..., string, asset
- Any **Foundational Dimensioned Type**: float3, int3, ..., matrix4d, quath
- Any Foundational Scalar Type Array: float[], int[], ..., string[], asset[]
- Any Foundational Dimensioned Type Array: float3[], int3[], ..., matrix4d[], quath[]
- Sentinel “Value Block”

Format implementations should keep the type of default in **agreement** with the typeName fields. Resolving the disagreement is format defined to support formats which may:

- Infer the typeName from the value
- Infer the precision of the value from the typeName
- Read the typeName without reading the value

Note that the field may store an explicit “Value Block” sentinel to block weaker opinions in composition without specifying a value. The sentinel block has a normative textual representation of None, but this is a distinct state. The sentinel value block agrees with any typeName, including opaque and opaque associated semantic aliases. The sentinel value block is the only value that agrees with opaque types.

The normative fallback value for default is an “empty state” which is not considered authoritative. This is the only type to have an unauthorable fallback value. Implementations have a lot of flexibility on how to implement this as it is not stored in any formats. They may choose to make this empty state a part of their VariantValue type, provide an optional return value, or not surface a fallback value for this field in their APIs.

Fallback value: Unauthorable Empty Sentinel

7.6.4.2.2 timeSamples: TimeSamples (Specialized Type)

A sequence of values of the same type ordered by discrete time samples.

- Key Type: double
- Value Type: A time sample may store a value of one of the following types:
 - Any **Foundational Scalar Type**: float, int, ..., string, asset
 - Any **Foundational Dimensioned Type**: float3, int3, ..., matrix4d, quath
 - Any Foundational Scalar Type Array: float[], int[], ..., string[], asset[]
 - Any Foundational Dimensioned Type Array: float3[], int3[], ..., matrix4d[], quath[]
 - Sentinel “Value Block”

Format implementations should keep the type of samples in agreement with the typeName fields. Resolving the disagreement is format defined to support formats which may:

- Infer the typeName from the time samples
- Infer the precision of time samples from the typeName

- Read the `typeName` or individual time samples without reading the whole time sample map

Note that the field may store an explicit “Value Block” sentinel to block weaker opinions in composition without specifying a value. The sentinel block has a normative textual representation of `None`. The sentinel value block agrees with any `typeName`.

Fallback value: `{}`

7.6.4.2.3 **connectionPaths: listop<ObjectPath> (Specialized Type)**

Stores a path representing a connection to another prim or property. Interpretation and evaluation of the connection is schema domain specified and not a part of value resolution. Attributes may have a value, have a connection, or both.

Targets of `connectionPaths` may be speculative.

Fallback value: `[]`

7.6.4.2.4 **spline: Spline**

Stores a spline representing a mapping function from time to value. If an inner loop is defined, negative numbers of `numPreLoops` and `numPostLoops` have no meaning and shall resolve to 0 if negative.

Discussion of value interpolation with regard to spline data can be found in the [Value Resolution section](#).

Value ranges:

Splines may store values that have the following types:

- `float`
- `double`
- `half`

Format implementations should keep the type of the spline values in agreement with the `typeName` field of the attribute storing the spline. Resolving disagreement is format defined to support formats which may:

- Infer the `typeName` from the spline data
- Infer the precision of the spline data from the `typeName`
- Reject data that cannot be coerced to the type defined by `typeName`

Fallback value: Spline instance with all default values

7.6.4.2.5 **allowedTokens: token[]**

For token valued attributes only, this field may be consulted to populate user interface options or to validate tokens.

This field should not be used for validation at the layer document level.

Fallback value: `[]`

7.6.4.3 Out of Scope Fields

These fields are reserved for inclusion in future specifications.

- colorSpace

7.6.4.4 Deprecated Fields

These fields are reserved though their usage is deprecated.

- connectionChildren

7.6.5 Relationship Spec

Relationship specs inherit all fields from property spec.

7.6.5.1 Value Fields

These fields are considered the property's primary values to be resolved. It is not required that targetPaths be authored. A relationship may be specified but not valued.

7.6.5.1.1 targetPaths: listop<ObjectPath> (Specialized Type)

Speculative list of targeted paths that be composed into the current layer stack.

Targets of targetPaths may be speculative.

Fallback value: []

7.6.5.2 Deprecated Fields

These fields are reserved though their usage is deprecated.

- relationshipTargetChildren
- noLoadHint

7.6.6 Variant Set Spec

7.6.6.1 Hierarchy Fields

These fields list the immediate children of the spec in the layer. These fields are non-speculative and should present the names of all child specs.

7.6.6.1.1 variantChildren: token[]

Array containing the persistent ordering of names of all child variant specs.

Fallback value: []

Value Ranges:

- Format storage should keep this field in agreement with spec storage. Handling child variant specs stored but not specified in variantChildren or values in variantChildren without a corresponding variant spec is format defined. This is so formats have the flexibility on how to keep them in sync or defer their reading.

7.6.7 Variant Specs

All prim spec fields are inherited by variant specs for the strict purpose of contributing opinions to composed prims. There are no fields that annotate or describe the variant itself.

7.7 References

1. OpenUSD API “SdfSchema” <https://openusd.org/release/glossary.html#usdglossary-path>.
2. Unicode Standard Annex #31 “*Unicode Identifiers and Syntax*”. <https://unicode.org/reports/tr31/>.

8 Paths

8.1 Introduction

A *path* represents an addressable location within a context. This context can be either a single layer, in which case the path refers to a specific spec within the context of that layer, or a composed stage, in which case the path refers to a composed scene object.

Examples of paths include [1]:

- Those used to represent and address prim specs within a layer or prims within a stage.
 - e.g., /Root/Child/Grandchild
- Those used to represent and address property specs within a layer or properties within a stage.
 - e.g., /Root/Child/Grandchild.visibility
- Those used to represent and address prim specs and property specs within a particular variant in a layer.
 - e.g., /Root/Child/Grandchild{size=1}/GreatGrandchild
 - e.g., /Parent/Child{color=r}.primvars:color.

Within this specification, paths can refer to a spec within a layer or scene object within a stage. Throughout the remainder of this section, the context should provide the semantics for that path, but for the sake of brevity, a path referring to a prim spec or prim will simply be referred to as a *prim path*. Similarly, a path referring to a property spec or property will simply be referred to as a *property path*.

Paths can be absolute or relative. An absolute path denotes a complete path from the root of the context (layer or stage) to the object being addressed. The root of the context is represented as /, and is referred to as the *absolute root path*. Relative paths denote paths that cannot be fully resolved until rooted elsewhere by anchoring the relative path to an absolute source. Relative paths can be represented in several different ways:

- A path starting with a prim name, indicating a child prim of the prim the path is rooted at.
 - e.g., Child/Grandchild (note that lack of the starting / which would indicate an absolute path)
- A path starting with a .., indicating the parent of the prim the path is rooted at.
 - e.g., ../Sibling/Child/Grandchild
- ., otherwise known as the *reflexive relative path* identifies the current prim the path is rooted at but may not prefix descendant prims.
- A path starting with . (but not exactly equal to .) indicates a property of the prim the path is rooted at.
 - e.g., .points

A relative path can only be interpreted once it has been *anchored* to an absolute path, resulting in the full addressable context of the object. A relative path *cannot* be made absolute by anchoring it to another path if by doing so would address ancestors of the

absolute root path. That is, the relative path `../../Descendant` cannot be anchored to `/Root` because doing so would cause the address to be interpreted on an ancestor of the absolute root path.

It is illegal for the parent specifier (`..`) to be placed in the middle of a path; it may only be used as the starting elements of the path. That is `../Sibling/Child/..` is an illegal path whereas `../../ParentSibling/Cousin` is a legal path.

An absolute path to any given prim can be formed by concatenating the identifiers of its parents together with the `/` character and appending it to the absolute root path.

Consider the following abstract hierarchy in a composed stage:

```
World
  Foo
    Baz
      Foobar
  Bar
    BarBaz
```

The prim `Foobar` can be addressed by the path `/World/Foo/Baz/Foobar`. Similarly, the prim `BarBaz` can be addressed by the path `/World/Bar/BarBaz`. A path anchored at `/World/Baz/BarBaz` to `Foobar` can be addressed by the path `../../Foo/Baz/Foobar`. Note this process is the same for addressing prim specs within the context of a layer.

Using the same composed stage, a property of a prim can be addressed by forming the path to the prim and concatenating that with the identifier of the property using a `.` character. Adding properties to our abstract hierarchy:

```
World
  Foo
    Baz
      Foobar
        foobarprop1
        foobarprop2
  Bar
    BarBaz
      barbazprop1
```

The property `foobarprop1` can be addressed by any of the given paths:

- `/World/Foo/Baz/Foobar.foobarprop1`
- `../../Foo/Baz/Foobar.foobarprop1` (anchored at `/World/Bar/BarBaz`)
- `.foobarprop1` (anchored at `/World/Foo/Baz/Foobar`)
- `Baz/Foobar.foobarprop1` (anchored at `/World/Foo`)

Note that a property is a terminal leaf in a path. The same process can be used to address a property spec within the context of a layer.

Finally, a path can address the content of a specific variant within the context of a layer. This is done using a variant selector of the form `{variantSetName=variantName}` rooted at

the prim spec the variant set is contained within. As an example, consider the following layer content (note this is different from above in that it does not represent a composed stage)

Let's make Foobar a prim spec of a new variant spec VarFoobar:

```
World
  Foo
    Baz
      VariantSet1
        VarFoobar
          Foobar
            foobarprop1
            foobarprop2
          VarFoobaz
            Foobaz
      Bar
        BarBaz
          barbazprop1
```

To then address the property spec foobarprop1, the path becomes:

```
/World/Foo/Baz{VariantSet1=VarFoobar}Foobar.foobarprop1
```

To address the prim spec Foobaz, the path would select the variant spec VarFoobaz, i.e.,

```
/World/Foo/Baz{VariantSet1=VarFoobaz}Foobaz.
```

Note that since variant specs may contain variant set specs, there can be a chain of selected variants specs. For example, consider this layer hierarchy:

```
World
  Foo
    Baz
      VariantSet1
        VarFoobar
          Foobar
            foobarprop1
            foobarprop2
          VarFoobaz
            Foobaz
      Bar
        VariantSet2
          VarBarBaz
            BarBaz
              barbazprop1
          VarBarFoo
            VariantSet3
              VarBarBar
                BarFoo
```

The prim spec BarFoo is addressable by the absolute path:

```
/World/Bar{VariantSet2=VarBarFoo}{VariantSet3=VarBarBar}BarFoo
```

Note that there is no / separator between a variant selector and another variant selector, nor between a variant selector and a prim spec. A variant selector may also be empty (e.g., of the form {x=}). This indicates that the selection is not the variant spec, but the variant set spec itself.

One difference to note above is that paths containing variant selectors are generally used only in the context of a layer. This is because when a stage is composed, a specific variant has been selected and the paths to those composed scene objects no longer use the variant selector. Consider the last example, and that for the composed stage we have selected VarFooBar and VarBarBaz as the selected variants. The composed stage then looks as follows:

```
World
  Foo
    Baz
      Foobar
        foobarprop1
        foobarprop2
  Bar
    BarBaz
      barbazprop1
```

Addressing the prim Foobar can be done using the path /World/Foo/Baz/Foobar. This is different from addressing the Foobar primspec in the layer, which was done via the path /World/Foo/Baz/{VariantSet1=VarFoobar}Foobar.

8.2 Element Ordering

- Normative ordering of elements should be defined as:
 - '_' shall be ordered before all ASCII letters and numbers
 - ASCII letters shall be case folded and then ordered according to their ASCII values
 - If ASCII letters differ only in case, upper case letters shall be ordered before lower case letters
 - ASCII numbers shall be ordered before ASCII letters
 - ASCII numbers shall be ordered as a group, skipping leading '0' values
 - If a group of ASCII numbers is equal between the two groups and differs only in the number of leading '0' values, the group with the lower number of leading '0' values shall be ordered first
 - Unicode characters shall be ordered according to their code points

For example, consider the following property names:

```
foobar
Foobar
_foobar
foo_bar
```



```
foo001bar001abc
foo001bar002abc
foo0001bar0002xyz
foo00001bar
a0
aü
ab
```

The rules above would order these names as:

```
_foobar
a0
ab
aü
foo_bar
Foobar
foobar
foo00001bar
foo001bar001abc
foo001bar002abc
foo0001bar0002xyz
```

When paths elements of different spec forms are compared, properties should be ordered before prims which should be ordered before variant sets. Variants cannot have siblings that aren't variants.

Sample ordering of children.

```
/a.b
/a/b
/a{b=}
```

8.3 Path Grammar

Textual representation of paths use a specific syntax which can be formally specified as a grammar using PEG [2] notation. This grammar specifies a set of production rules that describe a valid string representation of a path. Run-time interpretation of that path is based on the context in which the path is being used. The following formal specification serves as a reference for implementations to interpret the semantics behind the textual representation of a path.

Note: Textual representations of a layer enclose paths in an angle bracket pair (<>) but this is not considered part of the path grammar.

The *Path* production rule serves as the entry point for interpreting a path.

8.4 Conventions

Whitespace is declared explicitly as part of the production rules. Any spacing in the production rule that doesn't refer explicitly to a whitespace production is spacing to make reading the production rule easier.

- A short summary of PEG notation used:
 - <- represents a production; the specified rule on the left can be reduced by matching any rules on the right.
 - [x] indicates a character class; valid characters are any that fall into the specified class.
 - 'x' or "x" indicates a string, these are literal characters or strings present in the production.
 - / indicates a choice, and the parser must attempt to match each of the production rules in the order they are written until one of them succeeds, or they all fail (in which case the production cannot be satisfied).
 - () is used as a grouping mechanism such that the production rules inside are considered one production with respect to the PEG operator applied to it.
 - ? indicates the preceding production rule is an optional rule. The parser must interpret this as valid if the production is either not present at all or present and valid.
 - * indicates the preceding production rule may be present any number of times in sequence or may not be present at all. Colloquially, this may be thought of as a *zero or more* operator.
 - + indicates the preceding production rule must be present at least once, but can be present more than once in sequence. Colloquially, this may be thought of as a *one or more* operator.
 - ! indicates a look-ahead sequence that states the production rules following that operated on by ! are only valid if the sequence operated on by the ! is not present.
 - & indicates a look-ahead sequence that states the production rules following that operated on by & should only be evaluated if the sequence operated on by the & is present.

8.5 Grammar Definition

The textual representation of a path is defined by a simple set of production rules given below. Paths consist of elements that must be valid *identifiers*. The rules for identifiers are derived from identifier recommendations in the Unicode standard [3]. In particular, this grammar uses the [XID_Start] and [XID_Continue] character classes to form valid identifiers. This implies that the text representation of a path must be encoded (e.g., UTF-8) such that they can be interpreted as Unicode code points (comprising characters that do not exist in the private-use or non-character code point ranges). Since Unicode code point values for ASCII characters are equivalent to their ASCII value, no additional special handling is required.

All identifiers are case-sensitive, meaning that `myIdentifier`, `MyIdentifier`, and `myidentifier` refer to different identifiers. Identifier equivalence is code point (or byte) equivalence. The identifier `München` may encode `ü` as a single explicit code point or with two code points (an umlaut modifier and a `u`). These are two distinct identifiers in OpenUSD and runtimes must treat them as such.

Note: The `[XID_Start]` and `[XID_Continue]` character classes are referenced here and used in their default form from the Unicode database with no further modification. The specific set of code points that fall within these character classes can be obtained via the `DerivedCoreProperties.txt` file of the Unicode database [4]. The `[XID_Start]` and `[XID_Continue]` character classes can evolve over time; this specification is based on the character classes as they are derived in [5].

```
Space <- [U+0020] / [U+0009]
```

```
ForwardSlash <- '/'
```

```
Dot <- '.'
```

```
LeftCurlyBrace <- '{'
```

```
RightCurlyBrace <- '}'
```

```
Underscore <- '_'
```

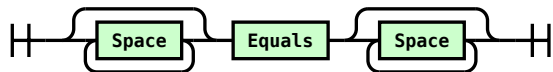
```
Equals <- '='
```

```
Colon <- ':'
```

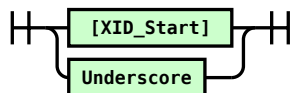
```
VerticalBar <- '|'
```

```
Hyphen <- '-'
```

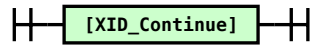
```
Assignment <- (Space)* Equals (Space)*
```



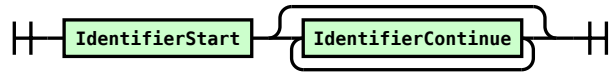
```
IdentifierStart <- [XID_Start] /  
Underscore
```



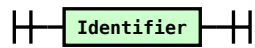
```
IdentifierContinue <- [XID_Continue]
```



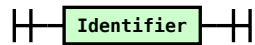
```
Identifier <- IdentifierStart (IdentifierContinue)*
```



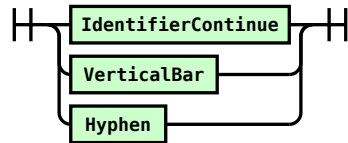
```
PrimName <- Identifier
```



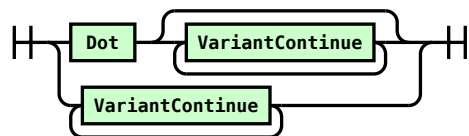
```
VariantSetName <- Identifier
```



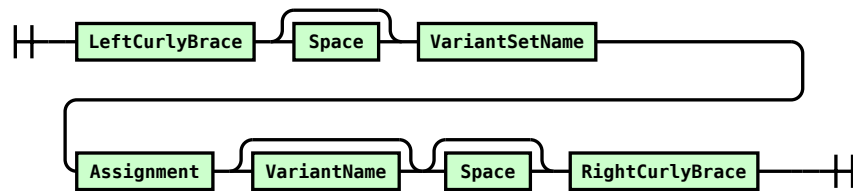
```
VariantContinue <- IdentifierContinue /  
                  VerticalBar /  
                  Hyphen
```



```
VariantName <- (Dot (VariantContinue)*) /  
              (VariantContinue)+
```



```
VariantSelection <- LeftCurlyBrace (Space)? VariantSetName ←  
                    Assignment (VariantName)? (Space)? RightCurlyBrace
```



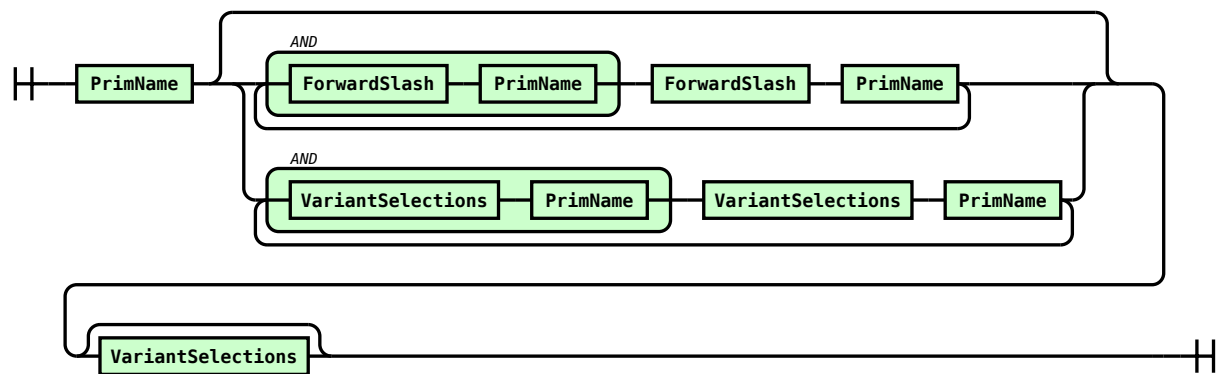
```
VariantSelections <- (VariantSelection)+
```



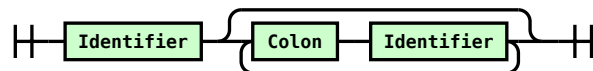
```

PrimElements <- PrimName (&(ForwardSlash PrimName) ForwardSlash PrimName)*
               (VariantSelections)? /
               PrimName (&(VariantSelections PrimName) VariantSelections PrimName)*
               (VariantSelections)?

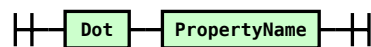
```



```
PropertyName <- Identifier (Colon Identifier)*
```



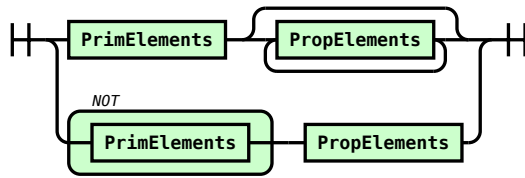
```
PropElements <- Dot PropertyName
```



```

PathElements <- PrimElements (PropElements)* /
               !(PrimElements) PropElements

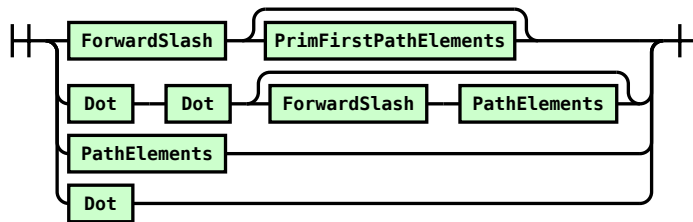
```



```
PrimFirstPathElements <- PrimElements (PropElements)?
```



```
Path <- ForwardSlash (PrimFirstPathElements)? /
      Dot Dot (ForwardSlash PathElements)? /
      PathElements /
      Dot
```



Note: Variant names are special in that they are allowed to start with anything in the [XID_Continue] character class, rather than restricted to the [XID_Start] character class. Be aware that the [XID_Continue] character class includes characters from the classes non-spacing marks (Mn), spacing combining marks (Mc), decimal numbers (Nd), and connector punctuation (Pc).

Reserved Characters

As this specification evolves, the Path grammar production rules may be changed to allow additional characters, particularly for enhancing what would be considered a valid *Identifier* in the grammar. However, in the absence of character escaping, there will always be a subset of characters that the Path grammar considers *reserved characters*, with the intent that an *Identifier* will likely never be valid if it contained one of these characters. These characters are defined by the following rule:

```
ReservedCharacters <- Dot /
                    ForwardSlash /
                    LeftCurlyBrace /
                    RightCurlyBrace /
                    Colon /
                    Equals
```

Note that under the [Compatibility with Legacy Content](#) section, the set of restricted characters contains additional characters not restricted by the normative specification.

8.6 Examples

The following paths are examples of paths that must be parsable by the grammar and a description of how runtimes should interpret them.

Valid Path	Description
/	A single “/” is used by runtimes to address the absolute root of the scene graph, often referred to as the “pseudo-root”. This is one of the few paths supported by the path grammar but isn’t encodable in path valued scene description.
.	Given the context of a prim path, a single “.” relatively identifies that prim.
.property	Given the context of a prim path, a “.property” relatively identifies a property of that prim.
..	Given the context of a prim, “..” relatively identifies the prim’s parent.
../..	Given the context of a prim, “../..” relatively identifies the parent of the prim’s parent.
../.points	Given the context of a prim, “../.points” identifies the points property of the prim’s parent
Descendant	Given the context of a prim, “Descendant” relatively identifies an direct child prim
/City{ selection = NewYork }	While path identifiers must not contain spaces, the grammar supports spaces and tabs in variant expressions. This path is considered to be equivalent to the path with all whitespace removed. “/City{selection=NewYork}”
/City/Street{selection=5thAvenue}	While path identifiers must not contain leading digits, variant selections may contain leading digits.
/City/Street{selection=}	A variant set without a variant selection identifies the variant set. This path identifies the selection variant set spec within the Street prim spec.

The following paths contain examples of paths that must be rejected by the grammar and an explanation as to why they're invalid.

Invalid Path	Description
./Descendant/Prim	Empty valued strings may be used in runtimes as sentinel values for an “empty path” or “no path”, but empty strings are not part of the path grammar. “.” cannot be followed by “/” to relatively reference descendants. Elide “./” to refer to descendants.
/New York/New York	Whitespace is not allowed in path identifiers
/abc/123	Leading digits are not allowed in path identifiers
/City/Street{=}	Variant selections may be empty, but variant set names may not be.
/abc{123=}	While variant selections may have leading digits, variant set names may not.
/Root/	A trailing slash is not valid
/Root//Child	Repeated slashes do not collapse.
/Prim/namespace:	The namespace separator must be medial
/Prim.:property	The namespace separator must be medial
/Prim.namespace::property	Repeated colons do not collapse.
/Prim.abc:123	Leading digits are not allowed in namespaced elements

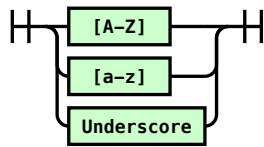
8.7 Compatibility with Legacy Content

The grammar defined above represents the normative specification for the Path grammar. However, as the OpenUSD ecosystem has evolved over the years, legacy content predating this specification exists. To provide maximum compatibility with this content, implementations have the option of supporting additional production rules for the Path grammar. The semantics of these rules are left unspecified; the additional production rules represent only a means to successfully parse legacy content for compatibility. Many of the production rules described in this section are additional rules but some normative rules are changed to support compatibility.

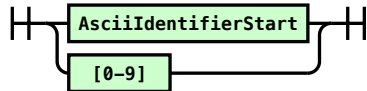
```
LeftBracket <- '['
```

```
RightBracket <- ']'
```

```
AsciiIdentifierStart <- [A-Z] /  
                      [a-z] /  
                      Underscore
```

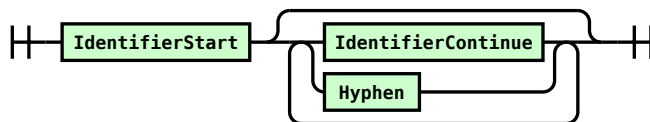
```
AsciiIdentifierContinue <- AsciiIdentifierStart /
                             [0-9]
```



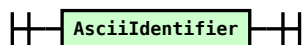
```
AsciiIdentifier <- AsciiIdentifierStart (AsciiIdentifierContinue)*
```



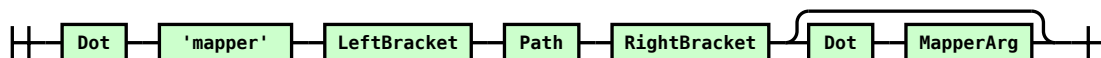
```
VariantSetName <- IdentifierStart (IdentifierContinue / Hyphen)*
```



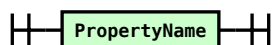
```
MapperArg <- AsciiIdentifier
```



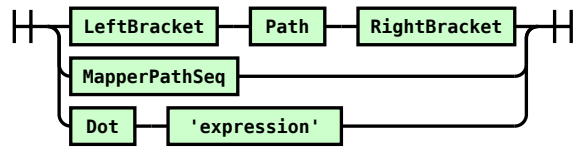
```
MapperPathSeq <- Dot 'mapper' LeftBracket Path RightBracket (Dot MapperArg)?
```



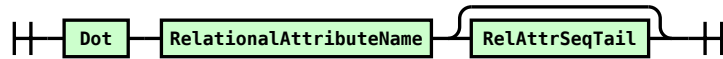
```
RelationalAttributeName <- PropertyName
```



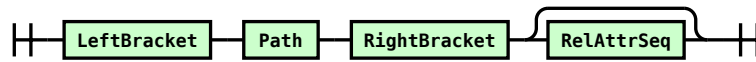
```
RelAttrSeqTail <- LeftBracket Path RightBracket /
                  MapperPathSeq /
                  Dot 'expression'
```



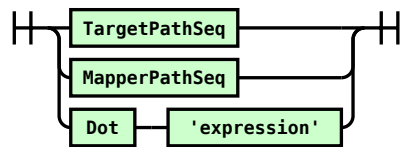
```
RelAttrSeq <- Dot RelationalAttributeName (RelAttrSeqTail)?
```



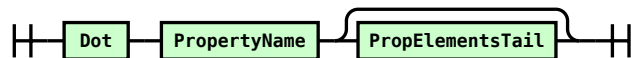
```
TargetPathSeq <- LeftBracket Path RightBracket (RelAttrSeq)?
```



```
PropElementsTail <- TargetPathSeq /
                    MapperPathSeq /
                    Dot 'expression'
```



```
PropElements <- Dot PropertyName (PropElementsTail)?
```



Additionally, under the compatibility layer, the set of reserved characters changes according to the following rule:

```
ReservedCharacters <- Dot /
                    ForwardSlash /
                    LeftCurlyBrace /
                    RightCurlyBrace /
                    Colon /
                    Equals /
                    LeftBracket /
                    RightBracket
```

8.8 Runtime Considerations

While the Path grammar specifies the syntax for how a path should be interpreted, and the examples above describe the semantics for how a path should be interpreted, there are a few additional runtime considerations worth mentioning:

- **Empty paths:** Empty paths are not valid productions under the Path grammar, but oftentimes it is helpful for the runtime to use empty strings to represent an “empty path” or “no path”.
- **Normalization:** Normalization is out of scope for the Path grammar. This means that Unicode characters that would normalize to equivalent under a particular normalization form (e.g., NKFD, etc.) are treated as distinct and hence, the paths containing those characters would be treated as distinct paths. For example, the strings “IX” (U+0049 U+0058) and “IX” (U+2168) are distinct strings when normalization forms are not applied.
- **Path Element Length:** There is no restriction on the length of an element in the Path grammar. While there may be a practical limit, runtime implementations should not assume anything about the maximum length of a Path element.

8.9 References

1. USD Terms and Concepts “Path” <https://openusd.org/release/glossary.html#usdglossary-path>
2. Ford, Bryan (January 2004) “*Parsing Expression Grammars: A Recognition Based Syntactic Foundation*”. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. pp. 111-122.
3. Unicode Standard Annex #31 “*Unicode Identifiers and Syntax*”. <https://unicode.org/reports/tr31/>.
4. Unicode Character Database. <https://unicode.org/ucd/>.
5. The Unicode Consortium. *The Unicode Standard, Version 15.1.0*, (South San Francisco, CA: The Unicode Consortium, 2023. ISBN 978-1-936213-33-7) <https://www.unicode.org/versions/Unicode15.1.0/>

9 Resource Interface

9.1 Scope

This section describes Universal Scene Description’s normative specification for resources or “assets”. Assets are often equivalent to files but can be generalized to refer to cloud or in-memory resources as well.

An asset value described by the [foundational data types](#) is the common storage type for resource identifiers in scene description.

The principles of the interface are specified here, but not the API.

9.2 Resource Identifiers

Universal Scene Description resources should be locatable using URI identifiers as specified by RFC 3986 [1].

9.2.1 URI Components

Section 3 of RFC 3986 breaks down a URI into four components:

- scheme
- hierarchy
- query
- fragment

A resource identifier specifies a “scheme” to identify a protocol for data interaction.

Schemes are case-insensitive and constrained to the grammar specified in Section 3.1 of RFC 3986.

9.2.2 Internationalized Resource Identifiers

RFC 3987 [2] describes support for Unicode characters in resource identifiers. Support for internationalized resource identifiers is delegated to schemes, protocols, and their implementations.

The scheme grammar for RFC 3987 does not deviate from RFC 3986 and is restricted to ASCII code points only.

9.2.3 Non-normative Identifiers

Schemes and protocol definition and implementations sometimes support additional identifier forms that do not strictly adhere to RFC 3986 or RFC 3987. Implementations should provide deference to schemes and protocols for what is considered allowable. It’s recommended that non-normative forms of identifiers be mappable to a normative URI. It’s recommended that operations on non-normative identifier forms produce equivalent results to the normative form.

An example of a non-normative identifier would be Windows file system paths. Windows file system paths can be converted to a normative file URI described by RFC 8089 [3]. RFC 8089's Appendix E and F describes several nonstandard syntax that implementations of file schemes sometimes support (i.e. \ as a separator).

9.3 Resource Protocols

Section 1.2.2 of RFC 3986 states that resource identification and interaction are separate. A resource identifier's scheme is used to identify the protocol for interfacing with a resource.

For performance, protocols for a resolved location should support parallel read access of data. Protocols may support updating or replacing a resource. Protocols may support a temporary detach to ensure parallel reads and writes do not conflict.

9.4 Relative Resource Identifiers

RFC 3986 defines syntax for both relative and absolute resource identifiers. Absolute resource identifiers for the purposes of this specification are those that have a scheme. Relative identifiers do not.

Relative resource identifiers require a base absolute resource identifier to be resolved. Section 5.2 of RFC 3986 defines the algorithm for resolving relative resource identifiers.

9.4.1 Anchored Relative Identifiers

Section 5.1 of RFC 3986 defines the set of potential sources for a base URI. Universal Scene Description treats `./` and `../` prefixed identifiers specially for the purpose of base acquisition when authored in a Universal Scene Description document. The base URI must be the authored document. No other bases should be considered.

The asset value

`./goodbye.txt`

authored in

`file:///hello/world.usda`

must resolve to

`file:///hello/goodbye.txt`

The asset value `goodbye.txt` authored in `file:///hello/world.usda` is not anchored. Implementations should not treat `goodbye.txt` and `./goodbye.txt` as equivalent when authored in a document.

9.4.2 Non-Anchored Relative Identifiers

Applications and implementations may define a set of base URIs for locating non-anchored relative identifiers. These are commonly structured as a "search path". A search path im-

plementation will commonly attempt to locate a resource at each base path, only reporting a failure after exhausting the set of paths.

Example search path:

```
[  
  "file:///lookup-location-1/",  
  "file:///lookup-location-2/",  
  "https:///remote-lookup-location/",  
]
```

Because non-anchored relative identifiers rely on application state, they are generally non-portable across facilities.

9.5 Resolving Identifiers to Locations

An identifier can be “resolved” to a resource location.

The algorithm for resolving an identifier is defined as:

- Apply relative resolution if necessary (RFC 3986 Section 5.2), providing special handling for anchored identifiers if necessary
- Normalize the identifier (RFC 3986 Section 6, RFC 3987 Section 5)
- Dispatch resource location to a protocol specified by the normalized identifier

It’s expected that if a resource has been located that it can be opened and data can be read. It’s acceptable for the resource location to be a “promise” that it should be available for opening and reading in the future.

A resource location should be representable as a URI, but it’s not required to be a persistent value. For example, an application may set up a temporary directory that stores mirrored assets from higher latency storage for the lifetime of the process. Applications could even garbage collect unused assets mid-process.

Resolving an identifier to a resource location may introduce a “change of scheme”. As an example, consider a strawman contextual scheme `json-file-lookup`. This scheme could specify that an application specified json file defines the location of assets. To resolve `json-file-lookup:AssetA`, a scheme resolver implementation could perform a lookup in a json file which resolves that identifier to a location on a filesystem `file:///resolved-usd-assets/AssetA.usd`.

9.6 Resolving Extensions

Extension based dispatch is fundamental to Universal Scene Description. Extension based dispatch allows configuring behavior without opening the asset.

Unless otherwise specified, an extension should be resolved by the following algorithm:

- Find the last element of the hierarchy component
- Tokenize it using `.` as the delimiter
- If there are no `.` characters, there is no extension

- Otherwise, the last element is the extension.

As an example, `file:///hello.world/data.v1.txt` should return `txt` as its extension.

A scheme with custom extension resolving might encode the extension in its query string.

For example, consider the following strawman URI: `texture-library:paint?extension=exr&color=red`.

The specification for `texture-library` could specify that the value of the extension element of the query determines the extension.

While “changing of scheme” may be common when resolving URI identifiers, changing of extensions is discouraged. Consider an identifier with a custom resolver scheme

`decal-library:stickers/rocketship.exr?quality=1k`

It would be confusing to users for that to resolve to `file:///textures/stickers/rocketship.1k.jpeg`.

9.7 Packaged Resources

Universal Scene Description reserves trailing `[]` to describe a resource internal to a package. For example,

`file:///asset-cache/AssetA.package[texture.exr]`

may be used to identify a resource inside another resource. A package format specification is responsible for describing how to locate and interface with nested resources. Packaged resources may be recursively defined:

`file:///asset-cache/AssetA.package[textures.zip[paint.exr]]`

9.8 Scheme Specifications

The specification does not require Universal Scene Description implementations to support any URI scheme but at least one must be supported in order to open a document. It’s recommended that implementations support the `file` scheme specified in RFC 8089. For environments that don’t have a file system, it is not required.

We recommend that implementations support the following scheme to describe in memory resources.

9.8.1 `usd-anon`

`usd-anon` may be used by implementations to describe in memory resources whose lifetime is tied to a process.

`usd-anon` resources should never be serialized into scene description.

`usd-anon` should never be used as a base path for anchoring or relative resolution.

9.9 Additional Notes

Single character URI schemes are not recommended as they can be confused with drive letters on Windows.

9.10 Security Considerations

Refer to Section 7 of RFC 3986 and Section 8 of RFC 3987 for more information about security considerations for URI and IRI identifiers respectively. Scheme specifications like file often provide additional security considerations (RFC 8089, Section 5) as well.

9.11 References

1. RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax. <https://datatracker.ietf.org/doc/html/rfc3986>.
2. RFC 3987 - Internationalized Resource Identifiers (IRIs). <https://datatracker.ietf.org/doc/html/rfc3987>.
3. RFC 8089 - The “file” URI Scheme. <https://datatracker.ietf.org/doc/html/rfc8089>.

10 Composition

10.1 Scope

This section introduces the algorithm for *composition*: the process of combining opinions for a prim path in multiple layers via a set of composition operators which themselves are specified in scene description. The composition algorithm is responsible for evaluating these operators and providing a “composed” view of a given prim.

10.2 Overview

The composition algorithm takes as input:

- The root layer from which composition should begin
- The path of a prim to compose
- Whether payload composition arcs are considered for this prim

The composition algorithm produces as output:

- A strength-ordered list of layer specs that contribute opinions to the prim specified in the input.

Executing the composition algorithm for a given prim is colloquially referred to as “composing the prim”.

10.2.1 Outline

```
compose_helper(root_layer, prim_path) -> [(LayerStack, PrimOrVariantPath)]:
    queue = []
    if prim_path == "/":
        queue.append((compose_layer_stack(root_layer), prim_path))
    else:
        # Recursively compose "ancestral opinions" from parent prims, then
        # append current prim's name to create initial list of opinions.
        ancestral_opinions = compose_helper(root_layer, prim_path.parent_path)
        for ancestral_layer_stack, ancestral_prim_path in ancestral_opinions:
            queue.append((ancestral_layer_stack,
                          ancestral_prim_path.append(prim_path.name)))

    while !queue.empty():
        layer_stack, prim_path = queue.pop()
        compose_arcs(layer_stack, prim_path, queue)

compose(root_layer, prim_path) -> [(Layer, PrimOrVariantPath)]:
    layer_stack_opinions = compose_helper(root_layer, prim_path)

    opinions = []
    for (layer_stack, path) in layer_stack_opinions:
        for layer in layer_stack:
```

```

        if layer.has_spec(path):
            opinions.append((layer, path))

    return opinions

```

10.3 Composition Operators

10.3.1 Sublayers

The sublayer composition arc superimposes scene description from multiple layers over one another. This composition arc forms an ordered list of layers called a “layer stack”, where the first layer in the list is considered the strongest layer and the last layer is considered the weakest.

Every layer stack begins with a given “root layer”, which is considered the strongest layer in the layer stack. Each layer in the layer stack may specify 0 or more additional layers to be added to the layer stack via the `subLayers` field, which provides a strong-to-weak ordered list of layer asset paths. These additional layers are considered weaker than the original layer. If the original layer was itself a sublayer, the additional layers are considered stronger than all weaker sibling sublayers of the original layer.

If a layer cannot be opened for a sublayer asset path (for example, if no layer exists at that path or the layer is malformed), it is a composition error and that sublayer is ignored.

If a specified sublayer would form a cycle when constructing the layer stack, it is a composition error and that sublayer is ignored.

10.3.1.1 Sublayer Time Offsets

Each sublayer may have an associated “layer offset” that applies a numerical offset and scale to all time values authored in that layer and any of its sublayers. By default, the offset is 0 and the scale is 1.0.

Sublayer offsets for a layer’s sublayers are stored in that layer’s `subLayerOffsets` field. The sublayer offset for a given sublayer is the layer offset in the list stored in the field at the same position that the sublayer’s asset path is stored in the `subLayers` field.

The effective layer offset for a layer in a layer stack is composed by taking the effective layer offset of the parent layer that included the layer in its sublayers and concatenating it with the layer’s sublayer offset. This is done by applying the scale of the parent layer to the offset and scale of the child and then adding the offset of the child to the result. Note that composition arcs like references may apply an additional time offset that concatenates with the layer offsets in the layer stack.

The composition algorithm is not responsible for applying layer offsets to time values. This is left to downstream clients, like value resolution.

If a sublayer offset has a scale less than or equal to 0, it is a composition error and that sublayer offset is treated as though it had the default offset and scale values.

10.3.1.2 Examples

root.usda

```
#usda 1.0
```

```
(
  subLayers = [
    @./sub_1.usda@ (offset = 5; scale = 2.0),
    @./sub_2.usda@
  ]
)
```

```
def "Root"
{
}
```

sub_1.usda

```
#usda 1.0
```

```
(
  subLayers = [
    @./sub_1_1.usda@ (offset = 10; scale = 3.0)
  ]
)
```

```
def "Root"
{
}
```

sub_1_1.usda

```
#usda 1.0
```

```
def "Root"
{
}
```

sub_2.usda

```
#usda 1.0
```

```
def "Root"
{
}
```

Opinions for /Root

Layer	Spec Path	Effective Layer Offset (offset, scale)
root.usda	/Root	(0.0, 1.0)
sub_1.usda	/Root	(5.0, 2.0)
sub_1_1.usda	/Root	(25.0, 6.0)
sub_2.usda	/Root	(0.0, 1.0)

10.3.2 Composition Arcs

Composition arcs are operators that specify opinions from a “source” layer stack and prim that should be included for the prim being composed.

These arcs are typically authored in scene description in list-op-valued fields. For example, a prim’s references composition arcs are stored in the `references` field. To compute all the arcs of a given type for a prim in a layer stack, the corresponding list-op from the weakest layer is applied to an empty list, then the list-op from the next strongest layer is applied to the result, and so on until all layers in the layer stack have been processed. This yields a list of 0 or more arcs of the given type where the first element is the strongest of those arcs and the last element is the weakest.

10.3.2.1 References

References add the composed opinions of a prim in a layer stack into the opinions of the referencing prim.

A ‘`reference`’ is an asset path, prim path, or a tuple of (layer asset path, prim path) where the layer asset path specifies the root layer of a layer stack and the prim path specifies the prim from which opinions should be composed. If no asset path is provided, the reference is considered an “internal” reference and the layer stack containing the reference is assumed. If no prim path is specified, the path to the prim specified by the `defaultPrim` field in the specified layer is assumed.

The list of references for a prim in a layer stack is computed by applying the list-op computation algorithm above using the `references` field.

For each reference, the layer stack is computed for the reference’s layer asset path. The composition algorithm is executed with that layer stack and the path of the prim indicated by the reference’s prim path to compute a list of opinions. These opinions are added to the opinions for the prim currently being composed.

If a layer stack cannot be computed for a reference’s layer asset path, it is a composition error and that reference is ignored.

If there are no specs in any of the layers of the referenced layer stack for the reference prim path, it is a composition error and that reference is ignored.

10.3.2.1.1 Namespace Mapping

References map the source namespace at and under the referenced prim to the target namespace where the reference was authored.

For example, given:

```
#usda 1.0
```

```
def "A" (  
    references = @./ref.usda@</B>  
)  
{  
}
```

the namespace mapping is [(/A, /B)].

If the reference is an internal reference, the namespace mapping also includes the identity mapping.

For example, given:

```
#usda 1.0
```

```
def "A" (  
    references = </B>  
)  
{  
}
```

the namespace mapping is [(/A, /B), (/, /)].

10.3.2.1.2 Reference Time Offsets

Each reference may have an associated “layer offset” similar to a sublayer offset. This offset applies to all layers in the referenced layer stack.

If a reference offset has a scale less than or equal to 0, it is a composition error and that offset is treated as though it had the default offset and scale values.

10.3.2.1.3 Examples

root.usda

```
#usda 1.0
```

```
def "LocalRef"  
{  
}
```

```
def "Root" (  
    references = [  
        "LocalRef"  
    ]  
)  
{  
}
```

```

        </LocalRef>,
        @./ref_1.usda@</Ref_1>,
        @./ref_2.usda@
    ]
)
{
}

ref_1.usda

#usda 1.0

def "Ref_1" (
    references = @./ref_1_1.usda@</Ref_1_1>
)
{
}

ref_1_1.usda

#usda 1.0

def "Ref_1_1"
{
}

ref_2.usda

#usda 1.0
(
    defaultPrim = "Ref_2"
)

def "Ref_2"
{
}

```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
root.usda	/LocalRef
ref_1.usda	/Ref_1
ref_1_1.usda	/Ref_1_1
ref_2.usda	/Ref_2

10.3.2.2 Payloads

Payloads are references that may either be composed (“loaded”) or ignored depending on the payload flag passed as input to the composition algorithm.

If the payload flag indicates that payloads should not be loaded, all payload composition arcs encountered during composition are ignored.

If the payload flag indicates that payloads should be loaded, all payload composition arcs are processed on the prim being composed and all ancestor prims that are composed during that process.

Otherwise, composing payloads is exactly the same as references, except that the list of payloads on a prim are computed using the `payload` field.

10.3.2.2.1 Namespace Mapping

Namespace mappings for loaded payloads are the same as for references.

10.3.2.2.2 Payload Time Offsets

Time offsets for loaded payloads behave the same as offsets on references. Time offsets for unloaded payloads are ignored.

10.3.2.2.3 Examples

root.usda

#usda 1.0

```
def "LocalPayload"
{
}

def "Root" (
  payload = [
    </LocalPayload>,
    @./payload_1.usda@</Payload_1>,
    @./payload_2.usda@
  ]
)
{
}
```

payload_1.usda

#usda 1.0

```
def "Payload_1" (
  payload = @./payload_1_1.usda@</Payload_1_1>
)
```

```

{
}

payload_1_1.usda

#usda 1.0

def "Payload_1_1"
{
}

payload_2.usda

#usda 1.0
(
    defaultPrim = "Payload_2"
)

def "Payload_2"
{
}

```

Opinions for /Root (payload excluded)

Layer	Spec Path
root.usda	/Root

Opinions for /Root (payload included)

Layer	Spec Path
root.usda	/Root
root.usda	/LocalPayload
payload_1.usda	/Payload_1
payload_1_1.usda	/Payload_1_1
payload_2.usda	/Payload_2

10.3.2.3 Inherits

The inherits composition arc adds composed opinions from a specified prim in the layer stack where the arc is introduced, as well as all layer stacks that were responsible for introducing that layer stack via other composition arcs.

The list of inherits for a prim in a layer stack is computed by applying the list-op computation algorithm above using the `inheritPaths` field.

For each inherit prim path, the composition algorithm is executed with the path and the layer stack containing the inherits. Those opinions are added to the opinions for the prim currently being composed.

Each inherit arc “implies” the presence of an inherit arc in all upstream layer stacks that introduced the original layer stack containing the authored inherit arc, up to and including the root layer stack. These “implied” inherit arcs behave as though an inherit arc was authored in these layer stacks on the prims that introduced those layer stacks. The prim path for the implied inherit arc is computed by applying namespace mappings from the original inherit prim path. Opinions from implied inherit arcs are computed like any other inherit arc and added to the opinions for the prim currently being composed.

If none of the layers in the layer stack where the inherits arc was authored contain specs for the inherit prim path, it is a composition error. It is *not* a composition error if there are no specs for any or all of the associated implied inherit arcs.

10.3.2.3.1 Namespace Mapping

Inherit arcs map the source namespace at and under the inherited prim to the target namespace where the inherit was authored, and all other paths to themselves.

For example, given:

```
#usda 1.0

def "A" (
    inherits = </B>
)
{
}
```

the namespace mapping is [(/A, /B), (/, /)].

10.3.2.3.2 Examples

root.usda

```
#usda 1.0

def "Root" (
    references = @./ref.usda@</Ref>
)
{
}
```

ref.usda

```
#usda 1.0

class "Inherit_1" (
```

```

    inherits = </Inherit_2>
{
}

class "Inherit_2"
{
}

def "Ref" (
    inherits = </Inherit_1>
)
{
}

```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
root.usda	/Inherit_1
root.usda	/Inherit_2
ref.usda	/Ref
ref.usda	/Inherit_1
ref.usda	/Inherit_2

10.3.2.4 Specializes

The specializes composition arc is similar to the inherits composition arc, but opinions added by this arc have a special strength ordering applied to them. See the [Strength Ordering](#) section for more details.

Composing opinions for specializes is exactly the same as inherits, except that the list of specializes are computed using the [specializes](#) field.

10.3.2.4.1 Namespace Mapping

Namespace mappings for specializes are the same as for inherits.

10.3.2.4.2 Examples

root.usda

#usda 1.0

```

def "Root" (
    references = @./ref.usda@</Ref>
)
{
}

```

ref.usda

#usda 1.0

```
class "Specialize_1" (  
    specializes = </Specialize_2>  
{  
}  
  
class "Specialize_2"  
{  
}  
  
def "Ref" (  
    specializes = </Specialize_1>  
)  
{  
}
```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
ref.usda	/Ref
root.usda	/Specialize_1
ref.usda	/Specialize_1
root.usda	/Specialize_2
ref.usda	/Specialize_2

10.3.2.5 Variants

The variants composition arc allows users to select which *variant* of scene description contained within a *variant set* should contribute opinions to a given prim.

The list of variant sets for a prim in a layer stack are computed by applying the list-op computation algorithm above using the `variantSetNames` field.

For each variant set, a corresponding variant selection must be computed to determine which of the variants in the set (if any) will be composed. If the computation returns no variant selection, the variant set is ignored. Otherwise, the composition algorithm is executed with the current layer stack and the prim variant selection path that addresses the selected variant in the variant set on the prim. Those opinions are added to the opinions for the prim currently being composed.

Evaluating variants must be deferred until all other composition arcs have been evaluated, so that all sources of opinions are available for computing the variant selection.

10.3.2.5.1 Computing Variant Selection

The variant selection is computed by examining the list of opinions for the current prim in strong-to-weak order and finding the strongest opinion for the `variantSelection` field whose dictionary value contains an entry whose key is the name of the variant set. The entry's value is the name of the variant selection.

If a variant selection was previously decided for a variant set in a layer stack, that selection must be used when composing the same variant set in other layer stacks.

10.3.2.5.2 Namespace Mapping

Variants do not remap namespace as they conceptually represent a “branch” of scene description in the same namespace as the containing prim. Thus, the namespace mapping for variants is the identity mapping `[/, /]`.

10.3.2.5.3 Examples

root.usda

#usda 1.0

```
def "Root" (  
    references = @./ref.usda@</Ref>  
    variants = {  
        string v = "x"  
    }  
)  
{  
}
```

ref.usda

#usda 1.0

```
def "Ref" (  
    variantSets = ["v"]  
    variants = {  
        string v = "y"  
    }  
)  
{  
    variantSet "v" = {  
        "x" {  
            def "X"  
            {  
            }  
        },  
        "y" {  
            def "Y"  
            {  
            }  
        }  
    }  
}
```

```

    {
    }
  }
}

```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
ref.usda	/Ref
ref.usda	/Ref{v=x}

10.3.2.6 Relocates

Relocates map opinions from the source layer stack (i.e. a layer stack included via another composition arc) to a new path in the current layer stack.

The relocates for a prim in a layer stack are computed by examining the `layerRelocates` field of each layer and finding the entry whose target path entry matches the prim's path. If such an entry is found, the composition algorithm is executed with the layer stack and the entry's source path to compute the opinions from the relocation source. Those opinions are added to the opinions for the prim currently being composed. All previously-computed ancestral opinions except those due to ancestral variant arcs are removed from the prim's list of opinions.

A relocated prim will still receive the same opinions from inherit arcs it would have had it not been relocated.

If any opinions are authored in a layer stack at a source path of a relocates statement in that layer stack, it is a composition error and those opinions are ignored.

If an entry in the `layerRelocates` field violates any of the following restrictions, it is a composition error and that entry is ignored.

- A relocate cannot apply to the pseudo-root or root prim paths.
- Relocate source and target paths must be prim paths.
- A relocate may not place a prim at the place of an existing ancestor. e.g. /Prim/Child/-GrandChild to /Prim/Child is not supported.
- A relocate may not place a target path below a source path. e.g. /Prim/Child cannot be moved to /Prim/Child/GrandChild.
- Each source path may only have one possible target, and each target may only have one possible source path.
- A target path must not be the same as a source path.
- If a relocate has "ancestral relocates" (e.g. an ancestor prim that has also been relocated), the relocate source path must use the ancestral relocated path. For example, if

you have /Root referencing /Ref, and /Ref also references /Ref2, if /Root/Ref is relocated to /Root/RefRelocated, any additional relocate that would use /Root/Ref/Ref2 as a source path must use the relocated path /Root/RefRelocated/Ref2.

10.3.2.6.1 Namespace Mapping

Composition arcs authored in layer stacks with relocates have an additional namespace mapping composed on top of their own namespace mappings. This additional mapping is computed by taking all relocates entries whose source path are prefixed by the path of the prim where the composition arc is authored, then adding entries to the mapping that map the source namespace at or under the relocates source path to the corresponding relocates target path.

For example, given:

root.usda

```
#usda 1.0
(
  relocates = {
    </A/B> : </A/C>,
    </Other/B> : </Other/C>
  }
)

def "A" (
  references = @./ref.usda@</Ref>
)
{
}
```

ref.usda

```
#usda 1.0

def "Ref"
{
  def "B"
  {
  }
}
```

the additional relocates namespace mapping for /A is **[(/A/B, /A/C)]**. When composing the reference arc authored on /A, this mapping composes with the standard reference namespace mapping **[(/Ref, /A)]** to produce the final namespace mapping for the reference arc **[(/Ref, /A), (/Ref/B,/A/C')]**.

10.3.2.6.2 Examples

root.usda

```

#usda 1.0
(
  relocates = {
    </Root/Child> : </Root/Relocated_Child>
  }
)

def "Root" (
  references = @./ref.usda@</Ref>
)
{
  over "Relocated_Child"
  {
  }
}

ref.usda

#usda 1.0

def "Ref"
{
  def "Child"
  {
  }
}

```

Opinions for /Root/Relocated_Child

Layer	Spec Path
root.usda	/Root/Relocated_Child
ref.usda	/Ref/Child

10.4 Strength Ordering

The strength order of two opinions X and Y for a given prim is computed by examining the layer stack/stacks that contain those opinions as well as the chain of layer stacks and composition arcs that introduced those layer stacks.

If X and Y have the same spec path but come from different layers in the same layer stack, then the stronger opinion is the one that comes from the stronger layer in the layer stack.

[see example 1](#)

Otherwise, the strength ordering is computed by examining the layer stacks and composition arcs that introduced the opinions.

If X was authored in a layer stack and Y was authored in some other layer stack that was introduced via a composition arc authored in X's layer stack, then X is stronger than Y. Said

another way, “local” opinions from a layer stack are stronger than all “remote” opinions that are introduced via composition arcs authored in that layer stack.

If X and Y were introduced by two different types of composition arcs authored in the same layer stack, the stronger opinion is the one introduced by the stronger composition arc type, which follow this order:

- Inherits
- Variants
- Relocates
- References
- Payloads
- Specializes (* see Specializes section below)

see [example 2](#)

This strength ordering of local and remote opinions is commonly referred to using the **LIVERPS** (pronounced “liver-peas”) mnemonic:

Local, **I**nherits, **V**ariants, **R**elocates, **R**eferences, **P**ayloads, **S**pecializes

If X and Y were introduced by the same type of composition arc authored in the same layer stack, their strength order is based on the following criteria:

- If X was introduced by a composition arc authored “deeper” in namespace than Y, then X is stronger, and vice versa. [see example 3](#)
- Otherwise, if X was introduced via an authored composition arc and Y was introduced via an implied composition arc, then X is stronger, and vice versa. [see example 4](#)
- Otherwise, X is stronger if the composition arc that introduced it was stronger than Y’s composition arc in the order computed when composing the list of composition arcs in that layer stack. [see example 5](#)

10.4.1 Specializes

Opinions introduced by specializes arcs follow a different strength ordering behavior. Specializes are ordered as the weakest of the composition arcs, but opinions introduced by specializes arcs are *globally* weaker than other opinions. Said another way, if prim A specializes prim B, then all opinions for prim A must be stronger than all opinions for prim B. This includes all implied opinions related to prim B and opinions from composition arcs that are introduced by prim B. [see example 6](#)

10.4.2 Examples

10.4.2.1 Example 1 (sublayer strength ordering)

root.usda

#usda 1.0

(


```

        subLayers = [
            @./sub_1.usda@,
            @./sub_2.usda@
        ]
    )

```

```

def "Root"
{
}

```

sub_1.usda

#usda 1.0

```

def "Root"
{
}

```

sub_2.usda

#usda 1.0

```

def "Root"
{
}

```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
sub_1.usda	/Root
sub_2.usda	/Root

10.4.2.2 Example 2 (composition arc ordering)

root.usda

#usda 1.0

```

def "Inherit"
{
}

```

```

def "Root" (
    references = @./ref.usda@</Ref>
)
{
    variantSet "v" = {

```

```

        "a" {
        }
    }
}

ref.usda

#usda 1.0

def "Inherit"
{
}

def "Ref" (
    inherits = </Inherit>
    variantSets = ["v"]
    variants = {
        string v = "a"
    }
)
{
    variantSet "v" = {
        "a" {
        }
    }
}

```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
root.usda	/Inherit
root.usda	/Root{v=a}
ref.usda	/Ref
ref.usda	/Inherit
ref.usda	/Ref{v=a}

10.4.2.3 Example 3 (namespace depth strength ordering)

```

root.usda

#usda 1.0

def "Root" (
    references = @./root_ref.usda@</RootRef>
)
{

```

```

    def "Child" (
      references = @./child_ref.usda@</ChildRef>
    )
    {
    }
  }
}

```

root_ref.usda

#usda 1.0

```

def "RootRef"
{
  def "Child"
  {
  }
}

```

child_ref.usda

#usda 1.0

```

def "ChildRef"
{
}

```

Opinions for /Root/Child

Layer	Spec Path
root.usda	/Root/Child
child_ref.usda	/ChildRef
root_ref.usda	/RootRef/Child

10.4.2.4 Example 4 (implied vs. authored ordering)

root.usda

#usda 1.0

```

class "RootClass"
{
}

```

```

class "RefClass"
{
}

```

```

def "Root" (

```

```

    inherits = </RootClass>
    references = @./ref.usda@</Ref>
)
{
}

```

ref.usda

#usda 1.0

```

class "RefClass"
{
}

```

```

def "Ref" (
    inherits = </RefClass>
)
{
}

```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
root.usda	/RootClass
root.usda	/RefClass
ref.usda	/Ref
ref.usda	/RefClass

10.4.2.5 Example 5 (sibling arc ordering)

root.usda

#usda 1.0

```

def "Ref1"
{
}

```

```

def "Ref2"
{
}

```

```

def "Root" (
    references = [
        </Ref1>,
        </Ref2>
    ]
)
{
}

```

```

    ]
{
}

```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
root.usda	/Ref1
root.usda	/Ref2

10.4.2.6 Example 6 (specializes strength ordering)

root.usda

#usda 1.0

```

def "Root" (
    references = @./ref.usda@</Ref>
)
{
}

```

ref.usda

#usda 1.0

```

class "Class_1" (
    specializes = </Class_2>
{
}

```

```

class "Class_2"
{
}

```

```

def "Ref" (
    specializes = </Class_1>
)
{
}

```

Opinions for /Root

Layer	Spec Path
root.usda	/Root
ref.usda	/Ref

Layer	Spec Path
root.usda	/Class_1
ref.usda	/Class_1
root.usda	/Class_2
ref.usda	/Class_2

For comparison, if the specializes arcs were all switched to inherits, the opinions would be:

Opinions for /Root

Layer	Spec Path
root.usda	/Root
root.usda	/Class_1
root.usda	/Class_2
ref.usda	/Ref
ref.usda	/Class_1
ref.usda	/Class_2

10.5 Namespace Mappings

Each composition arc has an associated “namespace mapping” that describes how to transform prim paths between a “source” namespace (where opinions are authored) and a “target” namespace (where the composition arc that composes in those opinions was authored). For example, if a prim /A has a reference arc to prim /B, then /B is the source namespace and /A is the target namespace.

Namespace mappings are bijections; as such they can be inverted, and multiple mappings can be composed together to create a single namespace mapping that maps the source namespace of the first mapping to the target namespace of the last.

Conceptually, a namespace mapping may be represented as a collection of (source path, target path) pairs. Each pair indicates that any paths in the source namespace at or under the source path can be transformed to the target namespace by replacing the source path prefix with the target path. The source and target path must both be prim paths, or both be the root path /. The latter case represents an identity mapping that maps a given path to itself.

If a path in the source namespace is not prefixed by any source paths in the namespace mapping, it cannot be transformed to the target namespace. This indicates that the object addressed by the path in the source namespace does not address an object in the target namespace.

10.5.1 Examples

With namespace mapping [(/A, /B)]:

Source Path	Target Path
/A	/B
/A/Child	/B/Child
/C	

With namespace mapping $[(/A, /B), (/ , /)]$:

Source Path	Target Path
/A	/B
/A/Child	/B/Child
/C	/C

10.6 Composition Errors

A “composition error” is an error in the specification of a composition operator in scene description. The composition algorithm does not terminate if it encounters a composition error when composing a prim; all other composition operators for that prim are evaluated and the computed opinions are returned as normal.

11 Stage Population

11.1 Scope

This section specifies the process of *stage population*, the process of forming a *stage* consisting of composed *prims* with the ability to fully resolve values taking into account all opinions in all layer specs contributing to the stage.

This section does not provide any specification for implementing this process; it is up to the reader to decide how best to do so for their unique requirements (e.g., fully populate and resolve, lazy populate, etc.).

11.2 The Stage

A *stage* represents a composed view of a scene graph from a layer root path down to each child path present in the composed layer. Each interior or leaf node in the scene graph represents a *prim*; the composed form of all prim specs providing opinions for the prim at a given path. A stage is defined by a *root layer*, which defines the layer spec providing the root of the scene graph. *Population* of a stage is the process of traversing each prim path reachable starting at the root of that layer, composing the opinions of each prim spec for that path, determining whether the composed prim should be active in the scene graph, and performing scene graph instancing to determine the final scene graph for the stage.

The content of each of the prims in the stage (e.g., attributes, metadata, variants, etc.) also require composition, and the process at which the final value of a metadata field is obtained is known as *value resolution*. In most cases, the final value of a metadata field is the strongest opinion of the composed opinions on that value contributing to the spec. In other cases, an additional value resolution process is performed on that composed data to return a final value. For example, some foundational data types support interpolation to produce values not explicitly specified in scene description.

11.3 Populating the Stage

Populating a stage starts with choosing a layer to serve as the root layer. Additionally, a *population mask* may be provided which may contain a set of absolute paths denoting exactly which objects to load into the scene graph. If provided, only the prims at those paths will be populated onto the stage. Ancestors of those paths *must* be populated to the stage as well. If no paths are provided, all objects reachable from the absolute root path will be populated. The population mask may also provide a flag indicating whether to load payload arcs, which can be passed as input to composition of [payloads](#).

Inputs: Root Layer, Optional Population Mask

Outputs: Composed Stage

```
populate_stage(root_layer, population_mask) -> stage:  
    queue = []
```



```

# create a scene object to represent the root path
layer = create_scene_object('/')
queue.enqueue(layer)

while !queue.empty():
    scene_object = queue.dequeue()

    # determine whether the scene object is active
    # if not, no need to consider it or any descendants
    if is_absolute_root_path(scene_object.path) or scene_object.value_resolve('active'):

        # compose the object
        scene_object.opinions = compose(root_layer, scene_object.path)

        # the scene object is active, determine if it's
        # instanceable - we only need to compose instanceables
        # once and then we can reuse that representation
        if !is_absolute_root_path(scene_object.path) and
            scene_object.value.resolve('instanceable') and
            has_composition_arc(scene_object):

            # the scene object is instanceable
            # if we already have a shared representation for it
            # we don't need to do anything, if we don't
            # we need to create the shared representation
            # and we need to populate the children the first time
            # which are then shared across all instances
            if not has_shared_representation(scene_object):
                create_shared_representation(scene_object)
                scene_object.children = populate_children(scene_object, population_mask, true)
            else:
                # we can skip everything else because we already processed the children
                # of the instance the first time
                continue
        else:
            scene_object.children = populate_children(scene_object, population_mask, false)

        # create scene objects for all children
        # then add those to the queue to compose
        for child in scene_object.children:
            queue.enqueue(child)

    return create_stage(layer)

populate_children(scene_object, population_mask, is_instanceable) -> child_scene_objects:
    # for each element in prim children, create a scene object
    # to represent the child, relocating if necessary

```

```

# and ensuring it is filtered in the population mask
child_scene_objects = []

# if it's instanceable, we need to filter out local opinions here
children = list_ordered_prim_children(scene_object)
if is_instanceable:
    children = filter_out_local_opinions(scene_object, children)

for child_name in children:
    child_path = scene_object.path.add_prim_path(child_name)

    # is it filtered by the population mask?
    # that is, does the population mask have content and
    # is this path either in the population mask or represents
    # an ancestor of a path in the population mask
    if population_mask and not population_mask.is_path_or_ancestor(child_path):
        continue

    # create a scene object for the child
    scene_object = create_scene_object(child_path)
    child_scene_objects.append(scene_object)

return child_scene_objects

```

Each population step requires visiting a path to:

- Create a scene object for that path
- Perform composition on that scene object
- Retrieve the children of that scene object
- Recursively visiting the child paths

These steps build the scene object hierarchy for the stage. Note that descendants are not considered for prims that are *inactive* (e.g., the **resolved value** of the **active** metadata field is false). Local opinions on descendants are also not considered for *instanced* prims (e.g., the **resolved value** of the **instanceable** metadata field is true) *and* the prim is composed of at least one composition arc (i.e., has at least one value in one or more of the **references**, **variantSets**, **payload**, **inheritPaths**, or **specializes** metadata fields).

11.3.1 Ordered Prim Children

The normative ordering of prim children requires merging **primChildren** from each contributing spec, resorting by **primOrder** after every merge. Specs are iterated on in reverse order. Relocates are applied once per layer stack, applying to all weaker layer stacks. Relocates that are “rename” operations within the same primitive must preserve the original order of the source primitive. Relocates that are “extend” operations which add additional children must **sort these relocated children** before merging. Relocates that “remove” or “rename” a child must prevent stronger layer stacks from introducing children with that name.

```

reorder(names, order) -> reordered:
    unordered_names = [n for n in names if n not in order]
    ordered_names = [o for o in order if o in names]
    return ordered_names + unordered_names

list_ordered_prim_children(scene_object) -> ordered_prim_children:
    prim_children = []
    # relocated_names are forbidden from being reused once relocated
    relocated_names = set()

    for layer_stack in reversed(scene_object.strength_ordered_layer_stacks):
        # Retrieve the relocates that affect the children of the scene object
        # removed_children and extended_children are token sets.
        # renamed_children is a token to token dictionary where the key is the source
        # (original name) and the value is the target (new name)
        removed_children, extended_children, renamed_children =
            layer_stack.relocated_children(scene_object)
        # remove children that have been relocated
        prim_children = [child for child in prim_children if child not in removed_children]
        # replace children that have been renamed due to a relocate
        prim_children = [child if child not in renamed_children else renamed_children[child]
            for child in prim_children]
        # append path element sorted children
        prim_children += path_element_sorted(extended_children)
        relocated_names = relocated_names.union(removed_children).union(renamed_children.keys())
        for spec in reversed(layer_stack.strength_ordered_specs):
            # When uniquifying a sequence, the first occurrence of a repeated element
            # must be preserved and all other occurrences deleted.
            prim_children = reorder(unique(prim_children + spec.prim_children),
                unique(spec.prim_order))
    return [child_name for child_name in prim_children if child_name not in relocated_names]

```

Each of these names are appended to the path of the parenting scene object as described [here](#) to form the path of each child to create a scene object for.

11.3.2 Ordered Property Children

The normative ordering of property children requires merging `propertyChildren` from each contributing spec with the composed `prim definition properties`, sorted by `path element ordering` and applying the strongest `propertyOrder` to the sorted children.

```

list_ordered_property_children(scene_object) -> ordered_property_children:
    property_children = set(scene_object.prim_definition.properties)
    for spec in scene_object.specs:
        property_children = property_children.union(spec.property_children)
    # See 'Ordered Prim Children' for reorder
    return reorder(path_element_sorted(property_children),
        scene_object strongest_authored_property_order)

```

Note: Unlike `primOrder`, only the strongest authored `propertyOrder` opinion affects the normative property order

11.3.3 Scene Graph Instancing

If the strongest opinion of the `instanceable` metadata field is `true`, the resulting prim is an instance prim so long as it has at least one composition arc. This has direct implication on the structure of the scene graph in that:

- An optimization should be made such that all instances of the same prim are composed only once. Overrides may not be specified on an instance prim.
- Local opinions are discarded, which also includes local opinions contributing to `primChildren`. This ensures that only the opinions brought in by the composition arc are used to determine the set of children, which are then shared across all instances of the same prim.

It is up to the implementation how to model the shared representation of instanced prims sharing the same arc.

11.4 Scene Graph Model Hierarchy

The model hierarchy is a contiguous subset of the prim hierarchy specified through the value of the `kind` metadata field of a prim. A prim can have one of the following well-defined kind values:

- `group`
- `assembly`
- `component`
- `subcomponent`

If the value of `kind` for a prim is `component`, the value of `kind` for the parent prim should be `assembly` or `group`. Similarly, if the value of `kind` for a prim is `assembly` or `group`, the value of `kind` for the parent prim should be `assembly` or `group` to produce a contiguous model hierarchy. Based on this, the traversal of the model hierarchy includes `component` but not descendants of the `component`.

For the purposes of model hierarchy traversal and continuity, `assembly` and `group` are equivalent. `assembly` implies a user defined point of interest while `group` does not. A fourth kind, `subcomponent`, is not considered part of the model hierarchy. It uses the `kind` field to tag important points of interest outside of the model hierarchy.

It is implementation dependent whether to validate model hierarchy rules. Other values of `kind` (including the empty string) are not included in the model hierarchy and should terminate continuity and traversal.

Implementations may provide customizable aliases for `assembly`, `group`, `component`, or `subcomponent`, but these shall not be considered generally portable.

11.5 Stage Queries

Prims can be queried via a set of predicate flags offering the ability to validate the usage of a prim as part of the scene traversal. As an example the application can query whether a prim is loaded or part of the model hierarchy. The following represents the list of flags that should be queryable on a Prim.

- Active - The resolved value of the active metadata field for the prim and all ancestors is true
- Loaded - Is the prim currently loaded or unloaded.
- Model Hierarchy - Is the prim part of the contiguous model hierarchy (i.e., the resolved value of kind is component, assembly, or group and all ancestor kind resolved values are assembly or group).
- Abstract - The resolved value of specifier for the prim and *any* ancestor is class
- Defined - The resolved value of specifier for the prim and *all* ancestors is **defining** (class or def)
- Instance - Is the prim marked as instanceable. See the discussion on **instancings**.

As an informative note, the query “active, loaded, defined and not abstract” generally provides the region of interest used in imaging and simulation. The term “concretely defined” may be used to mean “defined and not abstract”.

Stage traversal or queries should follow the preferred prim ordering as defined by the **path element order**.

12 Value Resolution

12.1 Scope

Value resolution is the process of resolving the value of a metadata field either at a specified time or when no time is specified. In most cases, the value of the metadata field is the value provided by the composed strongest opinion on the value of the field. However, there are a few cases that require additional computation. Although the process is similar for both attributes and metadata, there are subtle differences to how the values are resolved, in particular, metadata does not support time varying evaluations. This section discusses value resolution for metadata, attribute, and relationship values as well as the interpolation methods used where necessary to resolve those values.

12.2 Metadata Resolution

Metadata field value resolution is the simplest form of value resolution in that it is not time varying and retrieves the strongest opinion of a value determined by composition. That said, there are a few special cases that require a different process for value resolution.

12.2.1 specifier

Authored specifier values are considered *defining* (class, def) or *undefining* (over). Value resolution has special rules that take into the authored value of specifier on all composed specs.

- *Undefining* (over): All contributing opinions for the specifier field are over
- *Abstractly Defining* (class): The strongest *defining* opinion for specifier not from a direct inherit is class OR all of the prim's defining specifiers are class
- *Concretely Defining* (def): The strongest *defining* opinion for specifier is def OR the strongest *defining* opinion for specifier not from a direct inherit is def

While not a part of value resolution, the resolved value of ancestor's specifier field contribute to a prim's overall state. A composed prim is *defined* if a prim and all ancestor prims have a *defining* specifier (class or def). A composed prim is *concretely defined* if all ancestors are *concretely defining* (def).

More information on specifiers can be found in the [document data model](#).

12.2.2 typeName

In the case of both attributes and prims, the value of the typeName metadata field is determined by examining the [prim definition](#) and not the strongest opinion in the composed stack. Note this is different from other fields, that will examine the strongest opinion in the composed stack before consulting the prim definition.

12.2.3 variability

The attribute variability metadata field resolves similar to typeName in that the field is determined by examining the prim definition. If there is no value in the prim definition, value resolution will return the *weakest* opinion in the composition stack.

12.2.4 custom

The attribute custom metadata field has special value resolution that will examine all entries in the composition stack and return true if *any* of the opinions in the stack was authored to true. Note that if a property is a schema defined property, by definition it cannot be custom.

12.2.5 Dictionaries

Dictionaries have special combining rules detailed [here](#). Composition provides a strength ordering of dictionary value opinions and value resolution takes these opinions and combines them according to those combining rule, resulting in the value returned by value resolution.

The example below illustrates combining opinions on the customData dictionary.

```
def sphere "CueBall"(  
    customData = {  
        bool a = true  
    }  
)  
{  
  
    over "CueBall" (  
        customData = {  
            bool b = false  
        }  
    )  
    {  
        double radius = 2.0  
    }  
}
```

will result in value retrieved for customData being resolved as

```
customData = {  
    bool a = true  
    bool b = false  
}
```

12.2.6 List Op Resolution

Value resolution for a generic metadata field of type listop combines the entire stack of values for the field on a composed object. Implementations of value resolution must produce

a value congruent (but not equivalent) to the combined stack of opinions. Implementations may prune combination at the first explicit list operation or drop inert values in the stack as noted in the [specification for list operations](#) to produce congruent resolved list operation values.

Note: List op valued fields like references, payload, inheritPaths, specializes are inputs to composition and do not participate in standard metadata value resolution.

Note: Attribute connections and relationship targets value resolution follow the rules of generic metadata field list operation value resolution.

12.2.7 Layer Metadata

Layer metadata resolve values (including dictionaries) authored on the root layer spec and does not consider values authored on sublayers or other composition arcs. That is, these values do not participate in composition and no strength ordered opinion list is required to resolve.

12.2.8 Fallback

If no value is authored for a metadata field, value resolution should return the fallback value detailed for each field in the [data model](#).

12.3 Attribute Resolution

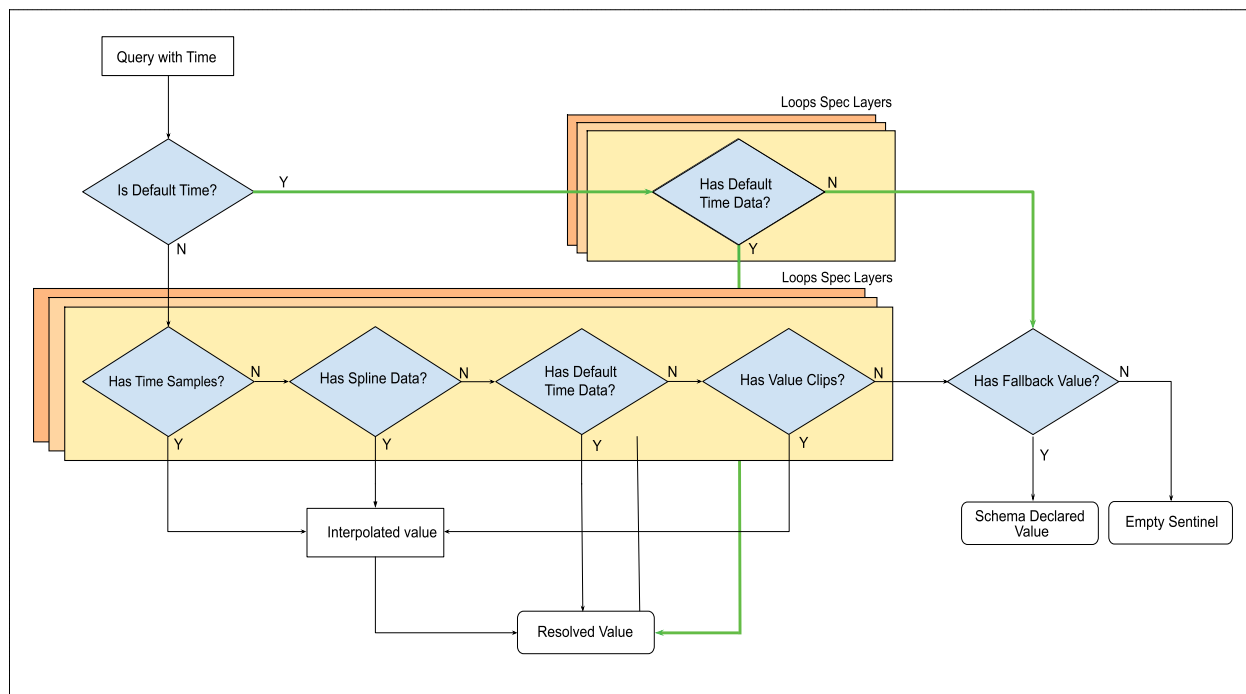


Figure 1: Attribute Value Resolution

Attribute values require more complex value resolution. As an attribute's value comes from the composition of layers that may contain opinions and authored values it is important that the correct value is queried from the appropriate layer. An attribute spec can hold three different metadata fields defining a value:

- `default`
- `timeSamples`
- `spline`

Furthermore, a prim containing an attribute can hold the following metadata fields that may affect that attribute's value:

- `clips`
- `'clipSets'`

Value clips allow time samples to be split into multiple layers; this is useful when dealing with heavy animated geometries like crowds or simulations and for scientific visualizations.

An attribute value can also be blocked. When a spec is marked as blocked in the opinion stack, weaker values need not be considered (and if the strongest opinion is blocked, the fallback value should be returned).

The layer stack is iterated from the strongest layer looking in the specs for authored values for the named attribute, either at a specified time or at the default time. The process will

stop the iteration once an appropriate authored value is found, which will be the authored value on the strongest available layer. If at the end of the iteration no authored value is found, the fallback value will be provided.

An attribute can have multiple datums authored on any given layer. In such a case the process first verifies if time is to be considered. If it is not then the default (non-time varying) value is evaluated by considering the strongest opinion of the default metadata field. If time based evaluation is considered then the strongest opinion of the following metadata fields are consulted (in the order given):

- timeSamples
- spline
- default
- clips

This happens on each spec contributing to composition. If no data is authored across the specs, then evaluation proceeds to the fallback value.

If a value at the default time is queried, the evaluation consults the default metadata field. If this field is unauthored, evaluation proceeds to the fallback value.

An example of a pseudo implementation for value resolution of time samples using the resulting composition is given below. In this case Held and Linear interpolation are supported. “Default” time is signified when time is defined as a “NaN” in the implementation and spec is the attribute spec to query.

```
'''
Interpolation of time samples.
In this example, timeSamples is a list of (time, value) pair
sorted by time:

ex: [(0, Float:6), (1, Float:8), (10, Float:-1)]

This example is using linear search for demonstration purpose,
ideally you would want to use more efficient search method
'''

'''
perform a basic linear interpolation.
See https://en.wikipedia.org/wiki/Linear\_interpolation
'''

def _linear_interpolate(self, lowertime, lowervalue, uppertime, uppervalue, time):
    interpolate_time = (time - lowertime) / (uppertime - lowertime)
    return lowervalue + interpolate_time * (uppervalue - lowervalue)

def _interpolate(self, timeSamples, time):
    # layout of timeSamples [0] is the time, and [1] is the Type
    # [(Time, Type:Value)]
    # TIME = 0 DATA = 1
```

```

minTime = timeSamples[0][self.TIME]
maxTime = timeSamples[-1][self.TIME]

# calculate pre and post values
if time < minTime:
    return timeSamples[0][self.DATA].value
if time > maxTime:
    return timeSamples[-1][self.DATA].value

# find where "time" intersects timeSamples
for time_data in timeSamples:

    if time_data[self.TIME] == time:
        return time_data[self.DATA].value

    if time_data[self.TIME] > time:
        prev = timeSamples[timeSamples.index(time_data) - 1]

        if self._interpolationType == InterpolationType.Held:
            return prev[self.DATA].value
        else:
            previous_time = prev[self.TIME]
            previous_value = prev[self.DATA].value
            current_time = time_data[self.TIME]
            current_value = time_data[self.DATA].value
            return self._linear_interpolate(previous_time, previous_value,
                                             current_time, current_value, time)

return None

def _process_time_samples (self, spec: Spec, time: float):
    if ( timeSamples := spec.fields.get("timeSamples")) is not None:
        timeCode = timeSamples.value
        timeCode.sort(key=lambda tc: tc[self.TIME])
        return self._interpolate(timeCode, time)

    return None

def get_value(self, spec: Spec, time: timeCode)->
    tuple[float, ValueResolutionProcess] | None:

    stage = self._stage

    # Do some basic validation
    if spec is None or spec.Form is not SpecForm.Attribute:

```

```

    return self.fallbackVal, ValueResolutionProcess.NoValue

# if default timeCode (non-time varying) is specified process
# default with fallback
if time.is_default():
    if (val := self._process_default(spec)) is not None:
        return val, ValueResolutionProcess.Default
    else:
        return self.fallbackVal, ValueResolutionProcess.Fallback

# process timeSamples
if (val := self._process_time_samples(spec)) is not None:
    return val, ValueResolutionProcess.TimeSamples

# process default
if (val := self._process_default(spec)) is not None:
    return val, ValueResolutionProcess.DefaultWithTime

# fallback
if (val := self._process_default(spec)) is not None:
    return val, ValueResolutionProcess.Fallback

return None

```

The authored time codes can be accessed from a spec with the following

```

if (time_samples := spec.fields.get("timeSamples")) is not None:
    timeCodes = time_samples.value

```

12.3.1 Default Values

The default value (“Has Default Time Data” in the diagram) is a specific datum on the specs that stores a value with **no specific time**.

To resolve this value, the specs for that attribute in each composed layer are queried for an authored default value. It will start at the strongest layer based on composition and iterate through the layers until an authored value is found. If no authored default value is found then what is known as the fallback value is returned. The fallback value for meta-data fields defined in the **core schema** are given in their respective sections.

If a value is queried at a specific time and no time samples have been authored, the query will ignore time and provide either the authored default value if found or a fallback value.

12.3.1.1 Authored Default Values

#usda 1.0

```
def Sphere "CueBall"
```

```
{
    double radius = 5
}
```

Extending the sample code, the default can be accessed with the following

```
def _process_default(self, spec: Spec):
    if (def_value := spec.fields.get("default")):
        return def_value.value
    return None
```

12.3.2 Time Based

When an attribute value is queried at a specific time, the query will take into account

1. Time Samples (from the timeSamples metadata field)
2. Spline Evaluation (from the spline metadata field)
3. Value clips (from the clips metadata field of the parent of the attribute)

The query will iterate all layers that have authored values on them starting at the strongest layer looking at each spec with the appropriate name for a time sample. Time Samples have higher priority over *Spline evaluation* or *Value clips*.

12.3.2.1 Layer Offset and Scale

References and sublayers can provide values to allow offsets in animation through the **Re-timing** specialized type. The time sample at a timeCode to be used will be computed from the optional offset and scale values provided when loading the composition.

```
#usda 1.0
(
    subLayers = [
        @./cueball.usda@ (offset = 10; scale = 0.5)
    ]
)
```

In this example if the current timeCode is 12 the time sample will be calculated $((12 * 0.5) + 10) = 16$, 16 being the new timeCode for sampling the animation in the sublayer.

It is important to note that scale **must be a positive none zero value**. Negative scale is particularly problematic as it results in time reversal and this causes problem when determining interpolation types for spline segments. For example, **inner loop interpolation requires** a start knot to be authored so reversing the spline will cause issues in evaluation of the inner loop.

Held interpolation is also affected by the reversal of time. **Held** interpolation uses the local time and does not take into account that the time samples have been reversed resulting in unexpected evaluations.

12.3.2.2 Time Samples

The following text example shows the radius of the sphere with time samples at (1, 5, 10) with associated values (0, 3.14, 6.28).

#usda 1.0

```
def Sphere "CueBall"
{
    double radius.timeSamples = {
        1:0,
        5:3.14,
        10:6.28,
    }
}
```

Example code to access the timeSamples

```
def _process_time_samples (self, spec: Spec, time: float):
    if timeSamples := spec.fields.get("timeSamples"):
        timeCode = timeSamples.value
        # sort based on time
        timeCode.sort(key=lambda tc: tc[self.TIME])
        return self._interpolate(timeCode, time)

    return None
```

timeCode will be in the form of a (time, value) pair in this case resulting in [(1, Float:0), (5, Float:3.14), (10, Float: 6.28)]

When the query finds a time sample on a layer for the attribute, interpolation is used to evaluate the correct value across the range of time samples at the time specified. For the specified time the intersection point is found between the two time codes and interpolation, either **Held** or **Linear** will be performed. Please see the [interpolation](#) section for more details.

12.3.3 Spline Evaluation

Using the active time, if the time falls inside the range of the authored definition the spline is evaluated using its basic form. If the time lies outside this range the evaluation needs to consider extrapolating loops or if authored inner loops to calculate the resulting value. More information can be found in [Interpolation: Extrapolating Loops](#) and [Interpolation: Inner loops](#)

12.3.4 Value Clips

Value clips allow the partitioning of time samples into multiple layers. Metadata defines how the frames map to the clip layers. Value clips specified within clip files are not subject

to composition, i.e., all data must be recorded directly, not inside variants or across references arcs. That is, while the clips and clipSets metadata fields compose as normal, the values inside the individual clip files do not compose.

12.3.4.1 Clip Sets

A “clip set” is a named group of value clips. The set of value clips along with sequencing and timing information and other value resolution behaviors are specified in the clipSets metadata field. The clip set definitions are stored in a dictionary-valued metadata field named clips. This allows users to define clip sets in various layers and have them compose together, or sparsely override metadata in clip sets non-destructively.

Clip sets may be defined using one of two possible forms of metadata.

- Explicit
- Template

Explicit metadata encodes the exact assets and sequence timings. Template metadata, on the other hand, authors a regex-style asset path template, and derives the explicit metadata when a stage is opened.

Template metadata is strictly less powerful than explicit metadata. It can’t achieve behaviors such as looping, reversing, or holding clip data, but it can provide a more compact and easy way to debug encoding for situations in which animation is broken up into a large number of regularly named files.

When both explicit and template clip metadata is authored, explicit will be chosen.

12.3.4.1.1 Common

Metadata common to both explicit and template formats.

12.3.4.1.1.1 primPath

A prim path that will be substituted for the stage prim’s path when querying data in the clips. For instance, if clip metadata is authored on prim /Prim_1, and primPath is /Prim the attribute /Prim_1.size would resolve to /Prim.size. This is the path that will be used to look for values in each clip.

12.3.4.1.1.2 manifestPath

An asset representing the path to a layer that contains an index of the attributes with time samples authored in the set of clips.

12.3.4.1.1.3 interpolateMissingClipsValue

A boolean flag indicating whether values for clips that do not have authored time samples for attributes in the manifest should be interpolated from surrounding clips. See [Interpolating missing value clips](#) for more information.

12.3.4.1.2 Explicit

Explicit metadata encodes exact assets and sequence timings.

12.3.4.1.2.1 assetPaths

An ordered list of asset paths to the clips holding time varying data.

12.3.4.1.2.2 active

A list of pairs of the form (stageTime, assetIndex) representing when a particular clip in assetPaths is active and should be considered during value resolution. See [Active time](#) for more details

12.3.4.1.2.3 times

A list of pairs of the form (stageTime, clipTime) representing the mapping from stage time to clip time, for whichever clip is active at the given stage time. See [Stage Times and Clip Times](#) for more details.

12.3.4.1.2.4 Explicit Metadata

Here is an example of an animated quad with a timeSample per frame using Explicit clips:

First the stage.usda containing the clips metadata for authored Plane:

```
#usda 1.0
(
    startTimeCode = 0
    endTimeCode = 2
)

def Xform "Geo" (
    clips = {
        dictionary default = {
            double2[] active = [(0, 0), (1, 1), (2, 2)]
            asset[] assetPaths = [./quad_1.usda@, ./quad_2.usda@, ./quad_3.usda@]
            asset manifestAssetPath = ./manifest.usda@
            string primPath = "/Geo"
            double2[] times = [(0, 1), (1, 2), (2, 3)]
        }
    }
)
{
    def Mesh "Plane"
    {
        float3[] extent = [(-1, -1, -0.1), (1, 1, 0.1)]
        int[] faceVertexCounts = [4]
        int[] faceVertexIndices = [0, 1, 3, 2]
        point3f[] points
```



```

    }
}

```

Clips are defined as metadata in the Geo transform. The clip files quad.1.usda, quad.2.usda, quad.3.usda have authored time samples 1,2 and 3 which get assigned to the stage time-codes 0,1,2 respectively. In this example we can also see that the Plane mesh doesn't have points. Points are defined in the clip layers quad.1.usda, quad.2.usda, quad.3.usda.

quad_1.usda

```

#usda 1.0
(
    defaultPrim = "Geo"
    metersPerUnit = 1
)

def Xform "Geo"
{
    def Mesh "Plane"
    {
        float3[] extent = [(-1, -1, -0.1), (1, 1, 0.1)]
        int[] faceVertexCounts = [4]
        int[] faceVertexIndices = [0, 1, 3, 2]
        point3f[] points.timeSamples = {
            1: [(-1, -1, 0), (1, -1, 0), (-1, 1, 0), (1, 1, 0)],
        }
    }
}

```

quad_2.usda

```

#usda 1.0
(
    defaultPrim = "Geo"
    metersPerUnit = 1
    upAxis = "Z"
)

def Xform "Geo"
{
    def Mesh "Plane"
    {
        float3[] extent = [(-1, -1, -0.1), (1, 1, 0.1)]
        int[] faceVertexCounts = [4]
        int[] faceVertexIndices = [0, 1, 3, 2]
        point3f[] points.timeSamples = {
            2: [(-1.2, -1.2, 0), (1, -1.2, -0.2), (-1, 1.2, 0), (1, 1, 0)],
        }
    }
}

```

```

    }
}

quad_3_usda

#usda 1.0
(
    defaultPrim = "Geo"
    metersPerUnit = 1
    upAxis = "Z"
)

def Xform "Geo"
{
    def Mesh "Plane"
    {
        float3[] extent = [(-1, -1, -0.1), (1, 1, 0.1)]
        int[] faceVertexCounts = [4]
        int[] faceVertexIndices = [0, 1, 3, 2]
        point3f[] points.timeSamples = {
            3: [(-1.3, -1.3, 0), (1.3, -1, -0.4), (-1, 1, 0), (1, 1, 0)],
        }
    }
}

```

Finally, the clip manifest specifies that the points attribute of the clips should be used.

manifest.usda

```

#usda 1.0

over "Geo"
{
    over "Plane"
    {
        point3f[] points

    }
}

```

12.3.4.1.3 Template Metadata

Template metadata based on a regex style authoring for asset paths.

12.3.4.1.3.1 templateAssetPath

A regex style template string representing the form of asset path names. This can be of two forms:

- path/clipname.###.usd
- path/clipname.###.###.usd

These represent integer stage times and sub-integer stage times respectively. In both cases the number of placeholders in each section is variable, and indicates how much padding to apply when looking for asset paths. Note that the format of this string is important. There must be exactly one or two groups of placeholders, and if there are two, they must be adjacent, separated by a dot.

12.3.4.1.3.2 **templateStartTime**

The double precision float first number to substitute into our template asset path. For example, given path/clipname.###.usd as a template string, and 12 as a template start time, the resulting asset path will have path/clipname.012.usd as its first element. If the template asset path represents integer frames and the start time has a fractional component, it will be truncated to an integer.

12.3.4.1.3.3 **templateEndTime**

The double precision float last number to substitute into our template string. If the template asset path represents integer frames and the end time has a fractional component, this component will be truncated to an integer.

12.3.4.1.3.4 **templateStride**

A double precision float number indicating the stride to be used for an increment when looking for files to resolve. For example, given a start time of 12, an end time of 25, a template string of path/clipname.#.usd, and a stride of 6, paths will be resolved as follows: path/clipname.12.usd, path/clipname.18.usd and path/clipname.24.usd.

12.3.4.1.3.5 **templateActiveOffset**

An optional double precision float number indicating the offset to be used when calculating the clip Active value. Given a start time of 101, an endTime of 103, a stride of 1, and an offset of 0.5, the following will be generated:

```
times = [(100.5,100.5), (101,101), (102,102), (103,103), (103.5,103.5)]
active = [(101.5, 0), (102.5, 1), (103.5, 2)]
```

Two additional clip time 'knots' must be generated on the ends of the clip Time array. This allows users to query time samples outside the start/end range based on the absolute value of their offset. In this case 100.5 and 103.5 are added.

Note that templateActiveOffset cannot exceed the absolute value of templateStride.

12.3.4.1.4 **Template Metadata Example**

Example of using a templated metadata extended from the example given for explicit.

```

def Xform "Geo" (
  clips = {
    dictionary default = {
      string templateAssetPath = [@./quad_###.usda@]
      double templateStartTime = 0
      double templateEndTime = 2
      double templateStride = 1
      asset manifestAssetPath = @./manifest.usda@
      string primPath = "/Geo"
    }
  }
)

```

In this example the clip files will be resolved to quad_001.usda, quad_002.usda and quad_003.usda based in the templateStartTime and templateEndTime. Once the files have been resolved the times and active metadata can be derived. For each time *t* specified in each derived assetPath, the times (t, t) will be authored; similarly, the active (t, n) will be authored, where *n* represents the index of the derived assetPath. If templateActiveOffset is specified, it will be applied to the derived times and active metadata.

12.3.4.2 Manifest

The manifest describes which attributes the value clip layers will provide. The manifest file allows value resolution to determine which attributes have clips available without having to query each clip file. In this case it defines that the points attributes of Plane will have values defined by a clip.

manifest.usda:

```

#usda 1.0

over "Geo"
{
  over "Plane"
  {
    point3f[] points
  }
}

```

12.3.4.3 Active Clips

The entries in the active metadata determines when a particular clip is active. Value resolution will retrieve values from the active clip for that given time.

A [stageTime, assetIndex] entry indicates that the clip in the assetPaths metadata at position assetIndex is active from time stageTime up to the stageTime of the next entry in the list. The first clip in the active metadata is also considered active for all *earlier* times, and the last clip is considered active for all *later* times.

As an example

```
double2[] active = [(0, 0), (1, 1), (2, 2)]
asset[] assetPaths = [ @./quad_1.usda@, @./quad_2.usda@, @./quad_3.usda@ ]
```

quad_1.usda, being the first clip to be loaded, will be active from time -inf to 1. Quad_2.usda is active from time 1 to 2 and quad3.usda, being the last clip, will be active from time 2 to +inf.

12.3.4.4 Stage time and Clip time

Conceptually, the (stageTime, clipTime) entries in the times metadata define a timing curve that specifies the time in the active clip to retrieve samples from when requesting an attribute's value at a given stage time. This timing curve is made up of linear segments whose endpoints are specified by the entries in times, sorted by stageTime.

For example, given this times metadata from the previous example:

```
double2[] times = [(0, 1), (1, 2), (2, 3)]
```

When an attribute value at time 0 is requested, value resolution will retrieve the time sample value authored at time 1 in the active clip, and at time 1 it will ask for the value authored at time 2 and so forth. These entries are the endpoints for a linear segment in the timing curve, so times between these entries will be linearly interpolated.

For example, requesting an attribute value at time 1.5 will cause value resolution to ask for the value authored at time 2.5 in the active clip.

The times metadata can be used to offset and scale animation from clips, providing flexibility in how they are applied to the stage.

12.3.4.5 Value Resolution of Clip Values

A clip set may provide values for attributes on the prim on which the clip set is defined and any attributes on descendants of that prim. It is important to note that value clips do not define attributes on a stage, they just provide values. If a clip set has values for an attribute but that attribute is not defined on the stage (for example, the attribute is not a built-in attribute of a schema), the clip set will not cause the attribute to come into existence.

The strength of data in a set of value clips is based on the anchor point. The anchor point is where the assetPaths or templateAssetPath is defined. The anchor point determines the strength of the value clip. The clip data is just weaker than the Local (L in **LIVERPS**) data of the anchoring layer. Clip data can be overridden by adding overrides to a stronger layer or in a local opinion, just as for any other kind of data.

During attribute value resolution, if clip sets are defined on the attribute's owning prim or its ancestors, value resolution will perform the following:

- Determine the path that will be consulted within clip layers when looking for values. The path will be constructed using the attribute's path within the local LayerStack,

with a prefix substitution based on the clip's `primPath` metadata. Composition arcs are ignored within clip files, i.e., all data must be recorded directly, not inside variants or across reference arcs.

- Find the strongest clip set that has the attribute at the above path declared in the manifest. This involves looking at the clip sets authored on the attribute's owning prim as well as that prim's ancestors.
- If no clip set is found, the process will iterate to the next spec layer.
- Query the clip set for the attribute value at the specified time. This "external" time will be mapped to the "internal" time of the clip set using the `times` metadata. The active clip will be opened and queried using this time.
- If an authored time sample at this time is found in the active clip, that is the final value. If there is no sample at that time, but there are other samples in the active time range of the clip, the final value will be interpolated from those samples. If values are missing, then the options defined in the manifest will define whether a fallback, sentinel, or interpolated value is returned.

To further detail the strength ordering of a Value clip consider the following example. In this example the manifest simply defines that the clip provides values for attributes "local" and "ref".

root.usda

```
def "Model_1" (  
  clips = {  
    dictionary default = {  
      asset[] assetPaths = [ @./clip.usda@ ]  
      asset manifestAssetPath = @./manifest.usda@  
      string primPath = "/Model"  
      double2[] active = [ (10.0, 0) ]  
    }  
  }  
  
  references = @./ref.usda@</Model>  
{  
  float local = 1.0  
  float local.timeSamples = {  
    5: 5.0,  
    10: 10.0,  
  }  
}
```

ref.usda

```
def "Model" (  
)
```

```

{
    float ref = 1.0
    float ref.timeSamples = {
        5: 5.0,
        10: 10.0,
    }
}

```

clip.usda

```

def "Model"
{
    float local = -100.0
    float local.timeSamples = {
        5: -5.0,
        10: -10.0
    }

    float ref = -100.0
    float ref.timeSamples = {
        5: -5.0,
        10: -10.0
    }
}

```

In this example the anchor is at the local layer. `assetPath` is defined in `Model`'s metadata. As mentioned, Value clips are slightly less strong than the local layer, meaning that requesting a value on attribute `local` at timecode 10 will result in a value of 10. In this case the value clip is ignored as values have been authored on a stronger layer, that is the local layer. However, if a value for `ref` was to be requested at time 10, the value returned will be -10. The value of "ref" from the clip data is resolved as it is stronger than the referenced layer.

To highlight the strength order further, the following example shows how both ancestor and descendants have to be visited in order to select the correct clip to use for value resolution. The "root.usda" contains a basic hierarchy based on a reference with an over on subgroup that authors a new clip for its attributes.

ref.usda

```

#usda 1.0

def "ModelGroup"
{
    float attr = 1.0
    float attr.timeSamples = {
        5: 5.0,
    }
}

```

```

def "Subgroup"
{
    float attr = 1.0
    float attr.timeSamples = {
        5: 5.0,
    }

    def "Model"
    {
        float attr = 1.0
        float attr.timeSamples = {
            5: 5.0,
        }
    }
}
}

root.usda

#usda 1.0

def "ModelGroup" (
    references = @./ref.usda@</ModelGroup>

    clips = {
        dictionary default = {
            asset[] assetPaths = [@./clip-group.usda@]
            string primPath = "/ModelGroup"
            double2[] active = [(0.0, 0)]
        }
    }
)
{
    over "Subgroup"
    {
        over "Model" (
            clips = {
                dictionary default = {
                    asset[] assetPaths = [@./clip-model.usda@]
                    string primPath = "/Model"
                    double2[] active = [(0.0, 0)]
                }
            }
        )
        {
        }
    }
}

```



```
}
```

With this given structure, querying a value in */ModelGroup* will yield values from the specified clip “clip-group.usda”. When the active prim is *ModelGroup/Subgroup/* any attribute value will be resolved by using the **ancestor’s** clip information as there are no locally authored opinions for prim *Subgroup*. However, when the prim is *ModelGroup/Subgroup/-Model* then the active clip will be “clip-model.usda”, i.e the descendant will override the ancestor, it is stronger. In this example, with clip sets authored on each prim, all value resolution is performed with value clips. The locally authored opinions are stronger than the reference hence the values authored in “ref.usda” are ignored.

12.3.4.6 Missing Values in Clip Set

A clip set has “gaps” if some of the value clips in the set do not contain authored time samples for an attribute that has been declared in the manifest.

By default, if a value clip does not contain time samples for an attribute, a time sample at the clip’s active time will be generated using the default value for the attribute authored in the clip manifest. If no default value has been authored, an empty sentinel value will be returned.

12.3.4.7 Interpolating Missing Values in Clip Set

Clip sets can optionally be interpolated by using the setting `interpolateMissingClipValues`. When enabled, if a query is made at a time when the clip set has a gap, and the attribute does not have a default value specified in the manifest, a forward and backwards search from the active clip at that time to find the nearest clips that contain authored time sample values will be performed. The final value will be interpolated from these time samples.

12.3.4.8 Jump Discontinuities

Jump discontinuities in the timing curve can be represented in the times metadata by authoring two entries with the same stage time, but different clip times. The clip time in the left-most entry is used for time mappings up to the specified stage time, while the clip time in the right-most entry is used for time mappings at that stage time and afterward.

For example, if there are two authored clips and the system requires animation from times 0 to 10 in the first clip followed by times 25 to 35 in the second clip. This could be specified with the active and times metadata like this:

```
double2[] active = [(0, 0), (10, 1)]
double2[] times = [(0, 0), (10, 10), (10, 25), (20, 35)]
```

A jump discontinuity has been specified at stage time 10. For times in the range [0, 10), resolution will retrieve values from the first clip at times [0, 10). For times in the range [10, 20], resolution will retrieve values from the second clip at times [25, 35]. Jump discontinuities can be used for looping, for example

```
asset[] assetPaths = [ @./clip_1.usd@ ]
double2[] active = [ (0, 0) ]
double2[] times = [ (0, 0), (25, 25), (25, 0), (50, 25) ]
```

This shows 25 frames of animation from a clip being looped from time 0 to 25, then 25 to 50.

12.3.4.9 Examples of using Active and Time metadata

12.3.4.9.1 Active clips

The defined assetPaths with assetIndex 0 and 1 will be used at times 0 and 10

```
asset[] assetPaths = [ @./clip1.usd@. clip_2.usd ]
double2[] active = [ (0,0), (10,1) ]
```

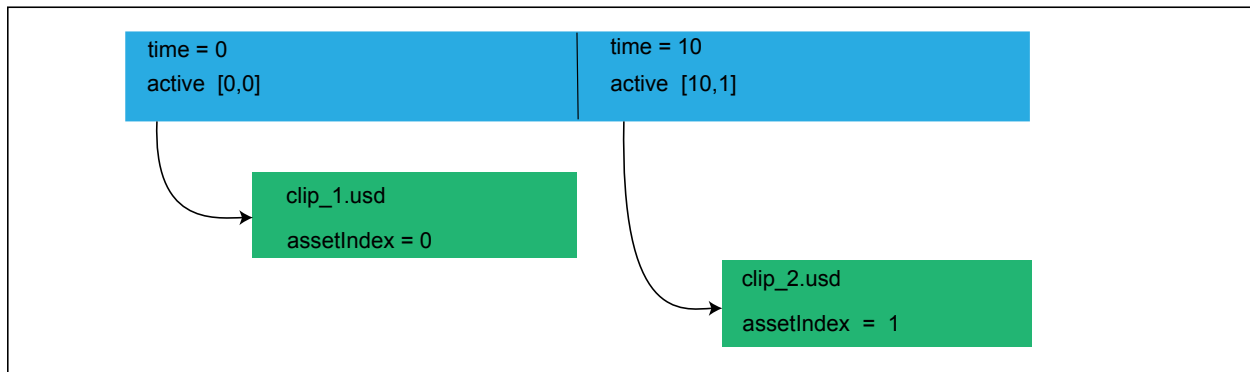


Figure 2: Active clip metadata

12.3.4.9.2 Clip times

Following on from the previous example, but now times is used to define a mapping between stage time and clip time. In this example, at stage time 0 the clip time is mapped to clip time 5.

```
asset[] assetPaths = [ @./clip1.usd@. clip_2.usd ]
double2[] active = [ (0,0), (10,1) ]
double2[] times = [ (0,5), (10,0) ]
```

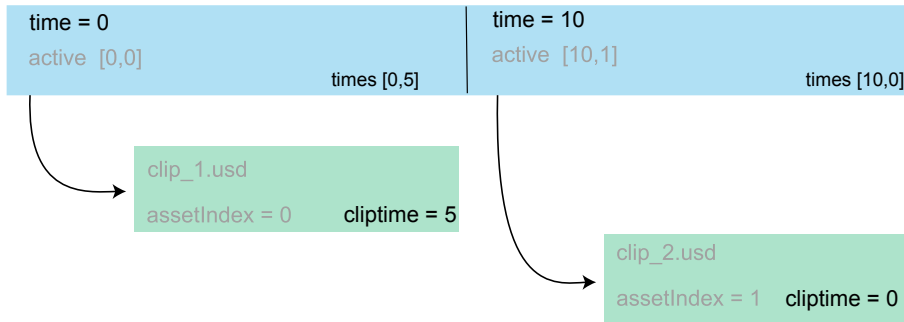


Figure 3: Clip time metadata

12.3.4.9.3 Clip Looping

```
asset[] assetPaths = [ @./clip1.usd@, clip_2.usd ]
double2[] active = [ (0,0), (10,1) ]
double2[] times = [ (10,0), (30,20), (30,0), (50,20) ]
```

In this example, extending again from the previous where looping takes place in clip_2.usd. The first iteration of the loop is between times [10,30] and then [30,50]. Value resolution at scene times samples 20 and 40 will result in the same value *V* being queried from the clip.

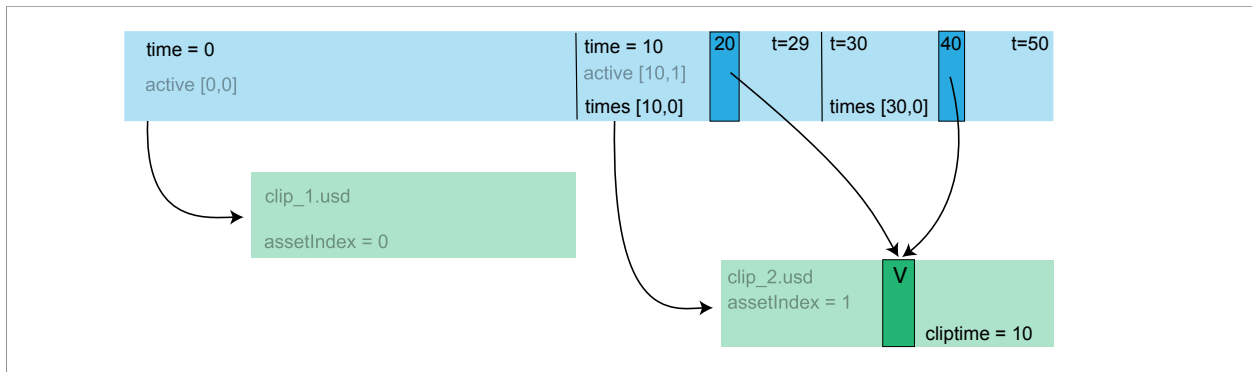


Figure 4: Looping with clip times

12.3.5 Fallback Values

- Each Prim has one or more **schemas** that define properties.
- The schema lists its properties, their types, and fallback values.
- When the stage is populated, all the prims are populated with properties coming from their schemas.
- If the user hasn't authored an opinion on a property, the fallback value provided via the **prim definition** is consulted according to **fallback value resolution**.

Note: The schemas of each prim present in the stage must be registered before the stage is populated, otherwise an accurate fallback value for an attribute defined via a schema cannot be provided if the schema is missing.

As an example, on loading a file that has a Sphere containing no authored attribute, the stage population will have to assume a fallback value. With the provided schema, or equivalent, the fallback value for, as an example, the “radius” can be queried.

12.3.5.1 Sphere Example

Here is an example of how we would define a new type called Sphere. The important part is that an attribute is defined with a fallback value.

```
class Sphere "Sphere" (
  doc = """Defines a sphere primitive."""
)
{
  double radius = 1 (
    doc = """Indicates the sphere's radius."""
  )
}
```

Now that we know how to define a Sphere, the prim can be created using the def specifier. This prim will contain all the properties declared in its schemas.

```
def Sphere "CueBall"
{
}
}
```

By using this definition, “CueBall” has an attribute called radius of type double with a fallback value of 1. Value resolution will result in a value for “CueBall.radius” of 1, as no value has been authored.

If no authored schema is available for the attribute then an empty sentinel value is returned.

12.3.6 Blocked Attributes

An attribute can also be blocked. When a spec is marked as blocked, weaker opinions on that spec are discarded. If the strongest opinion is marked as blocked, and a fallback value is available, the fallback value is returned. Otherwise, an empty sentinel value is returned instead. For more information on value blocks, see [value blocks](#). An example of validating the blocking capability with respect to accessing the default value can be seen below.

```
def _process_default(self, spec: Spec):
  """ Process the default value of the spec """
  if (def_value := spec.fields.get("default")) is not None:
```

```

        if(def_value.type == ValueType.ValueBlock):
            return self._process_fallback(spec)
        else:
            return def_value.value
    return None

```

In the following example the radius is blocked, any authored opinion on radius will be discarded and the fallback processed.

```

def Sphere "CueBall"
{
    double radius.timeSamples = {
        1: 2.0,
        2: 2.5
    }
}

def "RedBall" (
    references = </CueBall>
)
{
    double radius = None
}

```

Individual Time samples can also be blocked

#usda 1.0

```

def Sphere "CueBall"
{
    double radius.timeSamples = {
        1: None,
        2: 2.5
    }
}

```

Note that the per-timeSample-blocking ability does not allow us to sparsely override timeSamples, i.e. in the following example requesting a value at any time for “RedBall” will result in a ValueBlock resolving to fallback value if authored or a sentinel value being returned.

#usda 1.0

```

def Sphere "CueBall"
{
    double radius.timeSamples = {
        1: 2.0,
        2: 2.5
    }
}

```

```

    }
}

def "RedBall" (
    references = </CueBall>
)
{
    # This will force all time samples on RedBall to behave as Blocked
    double radius.timeSamples = {
        1: None,
    }
}

```

12.4 Relationships and Attribute Connections

Relationship targets are modeled as a list op of paths set on the `targetPaths` metadata field of the relationship spec.

While the semantics of what to do with a relationship target are up to the system querying it, the run-time shall provide the means to retrieve either the *raw* targets (i.e., those in the resolved listop of the `targetPaths` metadata field) or the *forwarded* targets, which recursively follow the relationship to the end of a target chain (a prim or an attribute).

Retrieving the raw targets is equivalent to retrieving the resolved value of the `targetPaths` metadata field. Retrieving the set of forwarded targets can be done through the following algorithm:

Input: A scene object containing a set of paths in a listop on the `targetPaths` metadata field

Output: A list of paths denoting the resolved forwarded relationship targets

```
get_forwarded_relationship_target_paths(scene_object) -> [resolved_paths]:
```

```

    # resolve the list_op to get the raw targets
    paths_listop = scene_object.resolve("targetPaths")
    resolved_targets = []
    for path in paths_listop:
        # if it's a prim or attribute path no forwarding required
        if is_prim_path(path) or is_attribute_path(path):
            resolved_targets.append(path)
        else:
            # if it's a relationship, the relationship has a listop
            # of targets, and we need to look at each of those and
            # forward
            sub_resolved_targets = get_relationship_target_paths(
                get_scene_object(path), true)
            resolved_targets += sub_resolved_targets

    return resolved_targets

```

For example, consider the following scene description:

```
#usda 1.0

def "foo" {
    rel myRel = [</foo/bar>, </baz.bazrel>]

    def "bar" {
    }
}

def "baz" {
    rel bazrel = [</foo/foobar>, </foo/foobar/barbaz>]
}
```

Using the algorithm above, the final set of targets for `/foo.myRel` would be `[/foo/bar, /foo/foobar, /foo/foobar/barbaz]`.

Attribute connections are modeled as a list of paths that relate an attribute to a set of targets and are set on the metadata field `connectionPaths` on the attribute spec.

Connection targets, unlike relationship targets are resolved to raw targets and not forwarded. Retrieving these targets is equivalent to retrieving the resolved value of the `connectionPaths` metadata field.

12.5 Interpolation

Attribute values can be associated with time samples and, with the appropriate interpolation flag set at the stage level, can be interpolated to retrieve the correct value at the queried time. At the stage level, two interpolation types can be specified:

- Held
- Linear

These values are used when time samples are authored. Spline evaluation is performed using its own specified interpolation method. If both time samples and splines are authored together, time samples will take precedence. When using time samples, if no interpolation method is specified at the stage level, the default interpolation is Linear. Splines interpolate is explicitly authored.

Specifying the stage level interpolation value is implementation defined.

In the example of the Cue ball prim, the radius of the Sphere can be animated using Held or Linear as defined. Each pair is a time and value matching the attribute type.

```
def Sphere "CueBall"
{
    double radius.timeSamples = {
        1:0,
        5:3.14,
```

```

    10:6.28
  }
}

```

An example of bezier spline based interpolation with curve interpolation at the knots with time (1,5,10) and Values (0,3.14,6.28) using default tangents. The end of time extrapolation is using Held.

```

def Sphere "sphere"
{
  double radius.spline = {
    bezier,
    pre: sloped(0),
    post: held(0),
    1: 0; pre (1, 1); post curve (1, 1),
    5: 3.14; pre (1, 1); post curve (1, 1),
    10: 6.28; pre (1, 1); post curve (1, 1),
  }
}

```

12.5.1 Held

The value is held to the nearest previous value at the authored time. This value will remain constant at subsequent time codes until another value is authored. For time codes before or after the authored time samples the value will remain “held” until the next authored sample.

For example:

```

def Sphere "CueBall"
{
  double radius.timeSamples = {
    100:10,
    200:20
  }
}

```

Queries from 100-199 will return “10”. Queries earlier than 100 will also return “10”. Queries from 200 and beyond will return “20”

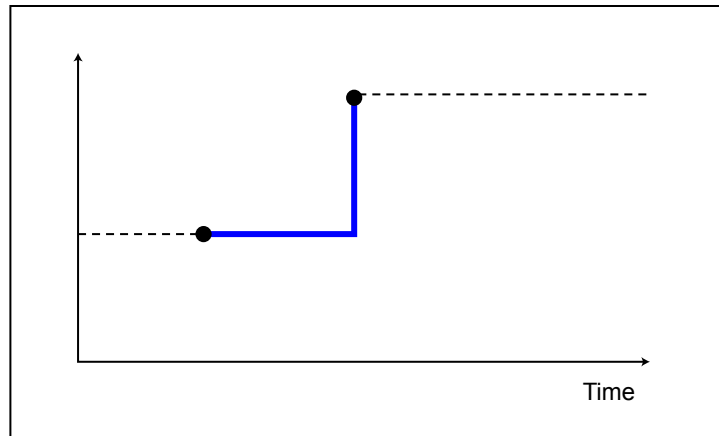


Figure 5: Held interpolation

12.5.2 Linear

Basic linear interpolation is supported for the following **types**:

- half
- float
- double
- timecode
- matrix2d
- matrix3d
- matrix4d
- half2 / float2 / double2
- half3 / float3 / double3
- half4 / float4 / double4
- quatd (via quaternion slerp)
- quatf (via quaternion slerp)
- quath (via quaternion slerp)

For time samples after the last authored sample, or for types that are not supported by linear, the computed value will behave as if Held.

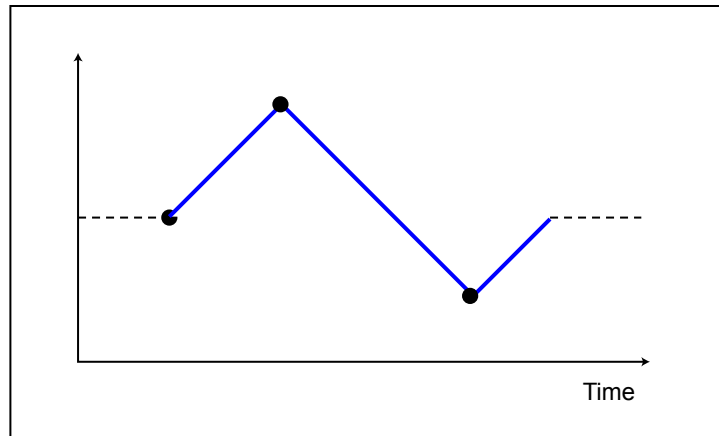


Figure 6: Linear Interpolation

12.5.3 Spline

Animation is defined using the following curve types:

- Bézier
- Hermite

Time based evaluation is performed across the length of the curve.

Splines support the following **data types**:

- half
- float
- double

12.5.3.1 Interpolation

Spline interpolation is how values are determined between the knots. There are four types of interpolation:

- value-block
- held
- linear
- curve
- If the interpolation type is value-block, there is explicitly no value between the knots.
- If the interpolation type is held, the value is held constant until the next knot.
- If the interpolation type is linear, the value linearly interpolated between the knot's values.
- If the interpolation type is curve, then value is computed based on the knot's values and tangents, and the spline's curve type.

Note: A spline has one curve type; it does not vary between curve segments.

A reference implementation of spline interpolation can be found in [AnimX](#).

12.5.3.2 Tangents

This specification uses the prefixes pre and post to define something happening before or after in the time dimension. Similar nomenclature uses the prefixes left and right, or in and out.

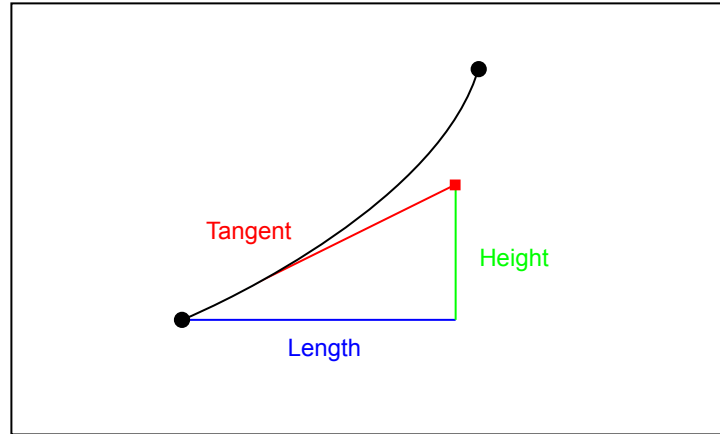


Figure 7: Tangent Definition

Tangents take the slope form. Slope is equal to the height divided by length. A positive slope increases in value as time increases, conversely a negative slope will decrease as time increases.

Other tangents forms exist such as using height instead of slope. Height based tangents typically differ by having negative values for upward sloping pre tangents. Conversion between tangent forms is left as an implementation detail.

Please reference [Cubic Splines Specialized Type](#) for more information.

12.5.3.3 Continuity

Although not stored as part of the spline or knot data, continuity classes C0 and C1 can be inferred. Discontinuity is supported through the usage of **dual valued knots**. Dual valued knots are defined with a **pre-value**, which is the value encountered when approaching from a lower time than the time value of the knot, and a **value** which is the value of the knot at the knot's time, and for interpolating beyond. Clearly, if the pre-value at the knot differs from the value, there is a discontinuity at that time.

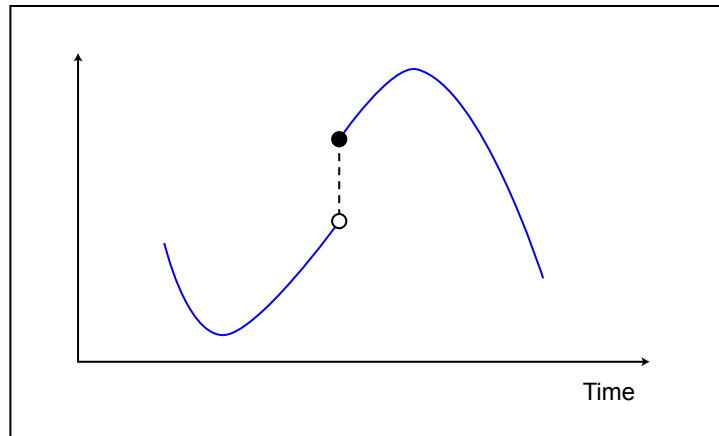


Figure 8: Continuity

Continuity	Continuous Value	Continuous Slope	Comment
Discontinuous	No	No	Dual Valued Knots, or Value-Blocks
C0	Yes	No	Tangents With Different Directions
C1	Yes	Yes	Identical Slopes

Note:

1. In the case of extrapolation with type `TsExtrapLoopReset`, a discontinuity potentially exists at the loop points.
2. Held values will also introduce discontinuity.
3. Inner loops can create discontinuity via the value offset.

12.5.3.4 Looping

Extrapolation determines the values before the first knot or after the last. Often this is used to perform looping.

12.5.3.4.1 Extrapolating loops

These use the entire spline (from first to last knot), and repeat it infinitely before and/or after the knots.

Extrapolation can take the following forms for both Pre and Post Extrapolation.

Type	Comment
block	No value
held	Constant Value

Type	Comment
linear	Linear Interpolation based on edged knots
sloped	Linear interpolation with specified slope
looprepeat	Knot curve repeated, curve is offset so ends meet.
loopreset	Curve repeated exactly, discontinuous joins.
looposcillate	Like Reset, but every other copy reversed

Extrapolating loops work across the whole authored spline. The default extrapolation type is Held.

12.5.3.4.1.1 Example loops

Syntax	Example	Comment
pre/post : sloped ,	pre : sloped(0)	0: Bezier 1. Hermite
pre/post: linear	post : linear	Linear interpolation
pre/post: held	pre : held	Held
pre/post loop	post : loop oscillate	Looping enabled, followed by loop type

12.5.3.4.1.2 Held Extrapolation

Retains the value of the end key. Held is the default extrapolation type. Typically used to maintain a constant animation, for example an object coming to rest and remaining at rest.

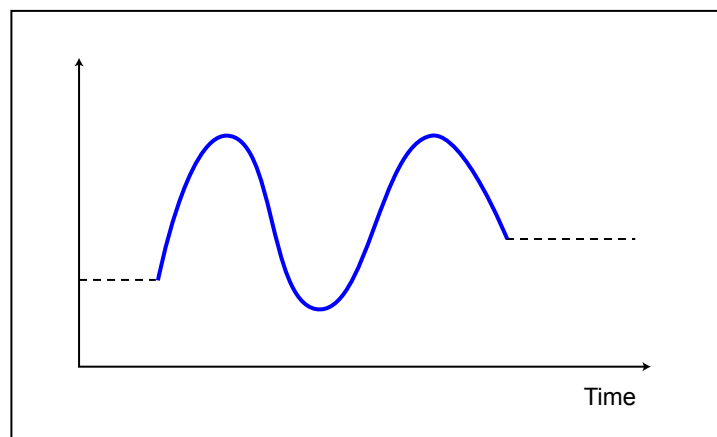


Figure 9: Held Extrapolation

12.5.3.4.1.3 Linear Extrapolation

Extrapolated value is calculated from the final tangent. It projects linearly to the end time frame. Commonly used to continue the animation, for example a car after a complex maneuver carries on moving formally in the same direction. linear Sloped Extrapolation is similar but a slope can be defined instead of being inferred by the first or last tangent.

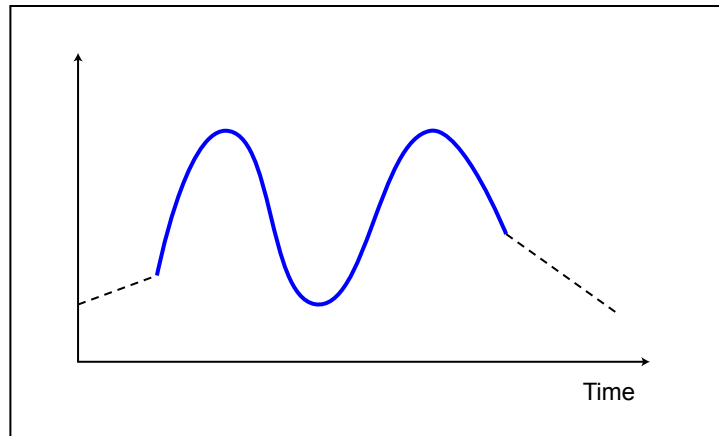


Figure 10: linear Extrapolation

12.5.3.4.1.4 Oscillate Extrapolation

The Oscillate setting repeats the animation curve by reversing its values, and therefore shape, with each cycle. Examples include animations where states switch back and forth, such as the motion of a piston.

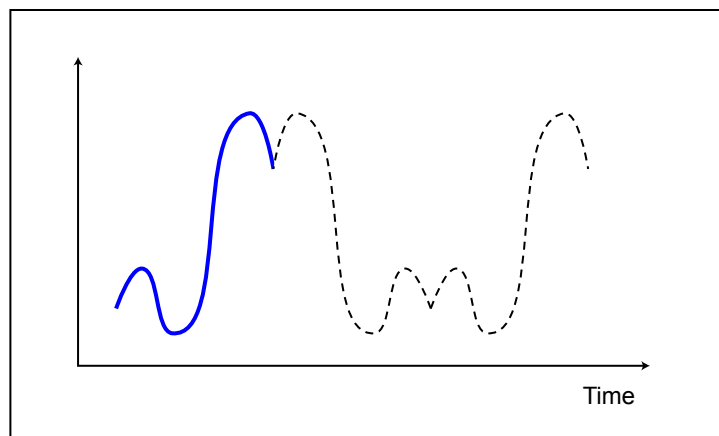


Figure 11: Oscillate

12.5.3.4.1.5 Repeat Extrapolation

Repeats the animation curve infinitely, except it appends the curve's last key's value to the value of the first key's original curve. This provides a seamless loop, but with an offset applied, thus producing a progressive loop. For example a character walking upstairs. The authored curve would be for one step on the stairs, the extrapolation will provide the result for each subsequent steps up the stairs.

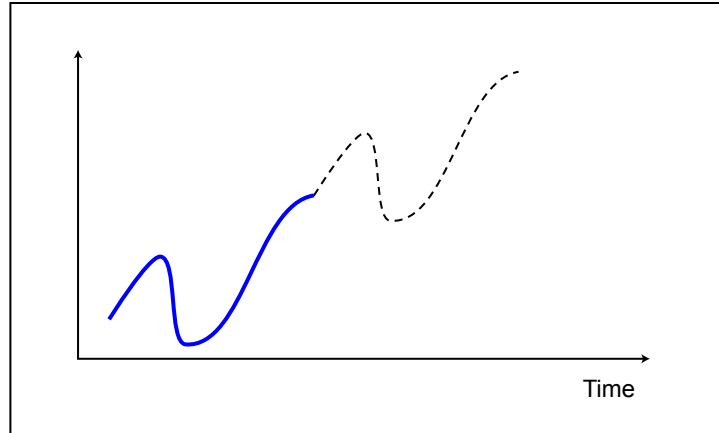


Figure 12: Repeat Extrapolation

12.5.3.4.1.6 Reset Extrapolation

Represent the curve as a repeat infinitely, potentially creating a discontinuity. These curves create a seamless loop, such as a walk cycle or a rotating fan.

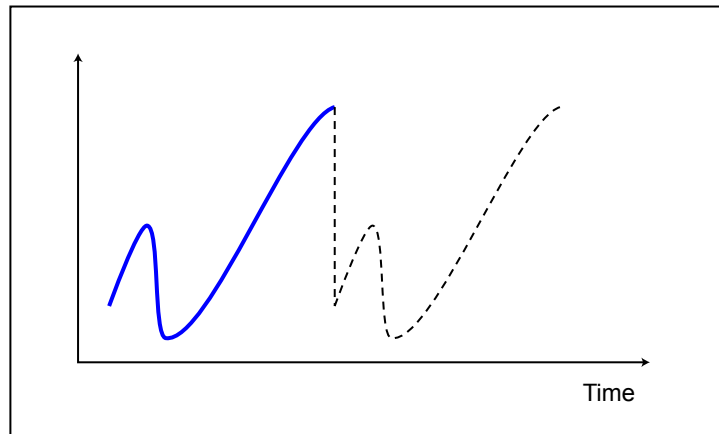


Figure 13: Reset Extrapolation

12.5.3.4.2 Inner loops

Inner loops specify a prototype region, which is repeated a finite number of times before and / or after the prototype region.

Inner loops are defined as part of a subset of the spline, known as a prototype.

Name	Type	Comment
protoStart	Time	Start knot of loop
protoEnd	Time	End time of loop
numPreLoops	Int32	Number of pre loops
numPostLoops	Int32	Number of pos loops

Name	Type	Comment
valueOffset	double	Value offset

protoStart must be located at a Knot, protoEnd specifies the end timeSample. Inner loops only support **Continue extrapolation method**

Prefix	Values	Example
loop	(start, end, preloops, postloops, valueOffset)	loop:(0, 60.0, 2, 2, 0)

Inner loops support the continue method.

12.5.3.4.2.1 Continue

The authored curve is copied from the protoStart to the end time sample, protoEnd. The start position must coincide with a knot. The curve is then pre and post copied based on the numPreLoops and numPostLoops values. As the end time sample doesn't always include a knot, the shape of the curve could be altered due to differing tangents. This method provides fine control over how extrapolation of the curve is performed.

Example	Description
loop:(10.0, 50.0, 0, 1, 0)	Authored curve, post copied once between timeSamples {10, 50}.

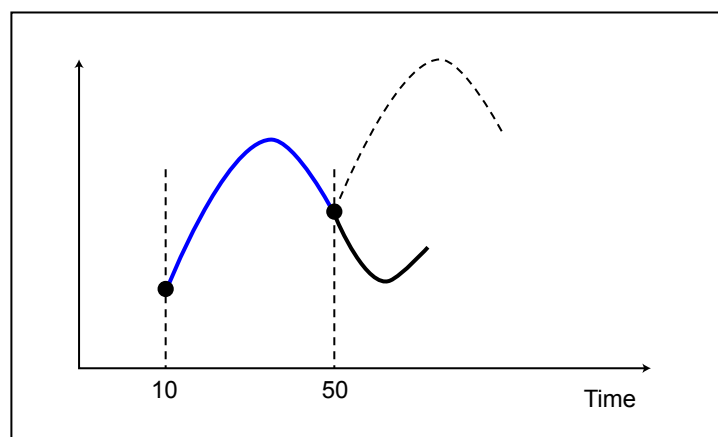


Figure 14: Inner Loop Example #1

Example	Description
loop:(10.0, 50.0, 0, 1, -5.0)	Authored curve, post copied once between timeSamples {10,50} with a -5 offset.

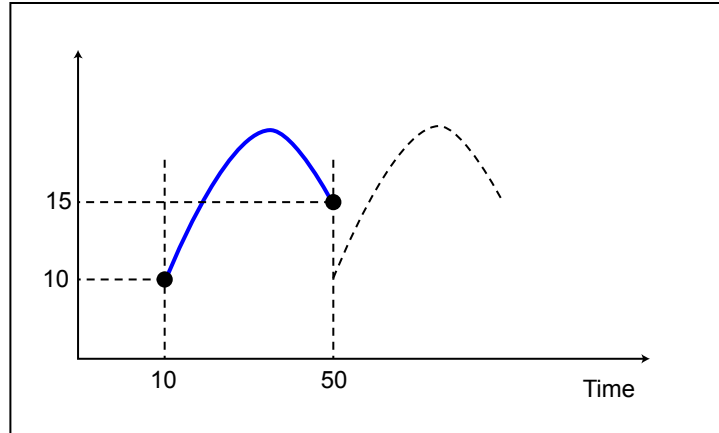


Figure 15: Inner loop #2

12.5.3.5 Non-monotonic Curves

A spline can use cubic Beziér curves for its shape. This shape represents a function that maps times to values. In the general case, it is possible for cubic Beziér curves to have shapes that are looped or S-shaped, that “double back” on themselves in time, giving a curve that has more than one value at some time or times. These shapes would violate the notion of a continuous single-valued function of time. These non-monotonic curves (sometimes referred to as regressive curves) refer to shapes where time values along the curve may decrease locally, resulting in non-monotonic progression. This usually happens when the extrapolated time range implied by a tangent precedes the previous knot’s time (for pre tangents), or exceeds the next knot’s time (for post tangents).

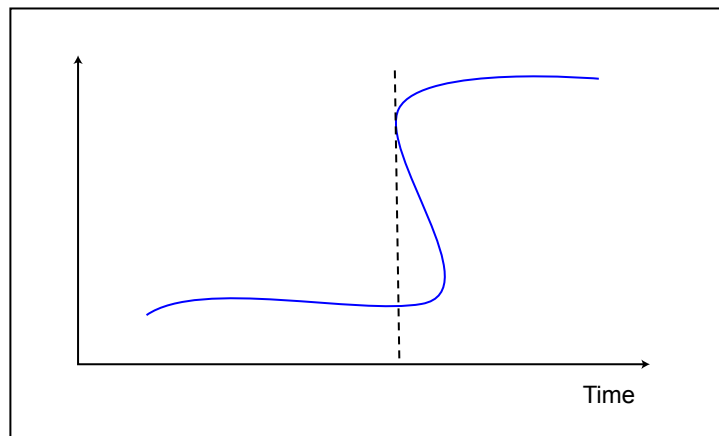


Figure 16: Non-monotonic Curves

This situation can be visualized as a segment of the spline bending back toward earlier time values, which creates a kind of temporal regression. Although rare, near-vertical tangents may be intentional to indicate extremely rapid changes; however, actual reversal in time (i.e., regression) produces evaluation errors or visual artifacts.

To prevent this, implementations should provide anti-regression functionality when authoring splines. Unless explicitly disabled, the anti-regression algorithm can modify the magnitude of authored tangents when necessary to ensure that the resulting curve is a single-valued function of time.

At evaluation time, if a spline is not a function of time, an implementation must correct it by shortening the tangents until the temporal regression is resolved. The tangents should be shrunk in proportion until the spline is a continuous function of time.

12.5.3.6 Custom Data

Custom data is supported on knots as specified [here](#).

13 Schemas

13.1 Scope

This section specifies the extension of well-known metadata fields to specs as well as ascribing a type and properties onto a prim via *schemas*. Both of these are used to drive downstream behaviors in the system.

This section does not specify the mechanism by which these extensions are defined or ingested into the runtime system and leaves that up to the specific implementation.

13.2 Extension Metadata Fields

Schema domains may define semantics for fields introduced as *extension metadata*. However, note that properties are preferred and recommended.

13.2.1 Core Metadata Extensions

Four additional metadata fields are added as extensions to the core schema definitions. These fields affect *stage population*, *composition*, and *value resolution*.

13.2.1.1 fallbackPrimTypes: dictionary

Map of schema names that, if not registered, should be populated as another more widely available schema. The value type of each entry of the dictionary should be a token array. It should contain no other entries. Both keys and elements in the token array should be valid type names, but this shouldn't be considered a value restriction. The values in each token array define a preference-ordered list of schema types to substitute, if available.

Applies To: Layer Specs

Fallback Value: {}

Example:

```
{  
  "DepartmentSphere" : ["FacilitySphere", "Sphere"]  
}
```

Fallback prim type semantics should not be applied at the layer level and instead shall be interpreted on the composed prim.

13.2.1.2 apiSchemas: listop<token>

Identifiers of applied schemas, or identifier/instance pairs of multiple applied schemas separated by a :.

Note that while *apiSchemas*'s elements are intended to follow the identifier or identifier/instance pair pattern, this is not a formal requirement of the layer document model.

Applies To: Prim Specs

Fallback Value: []

Applied schemas have no semantic meaning at the layer level and instead shall be interpreted on the composed prim.

13.2.1.2.1 clips: dictionary

Defines the clip sets on the prim. See [value clips](#).

Applies to: Prim Specs

Fallback Value: {}

13.2.1.2.2 clipSets: listop<string>

Listop defining the sets of clips used for a prim. See [value clips](#)

Applies to: Prim Specs

Fallback Value: []

13.3 Schema Types

Schemas are structured types that ascribe properties to a composed prim, which are used to drive downstream behaviors. These types can be authored onto prim specs to give that spec an opinion about what structured data the composed prim will receive. While authoring opinions about schemas are done on the prim spec, the resulting composed value shall only be interpreted by the runtime system on the composed prim. If a composed prim has an authored schema type, it will be imbued with additional properties defined by those authored schemas.

A schema can have one of two types:

- Typed
- Applied

A prim spec may have at most one typed schema authored onto it, but may have as many applied schemas authored onto it as it needs to fulfill its intended functionality in the stage. In this way, the composed prim shall have at most one typed schema and as many applied schemas associated to it as dictated by composing these authored fields.

13.3.1 Typed Schemas

A *typed schema* defines the concrete type of a composed prim. Opinions about this type are authored to the typeName [metadata field](#). This value is resolved via composition to result in a composed prim that either has a type or is typeless.

A typed schema definition must contain:

- A name for the schema
- A definition of zero or more properties

These properties are considered part of the **composed prim definition** for a prim of that type.

Typed schemas may be *abstract* or *concrete*. An abstract typed schema, like a concrete schema, optionally provides the definition of a set of properties, but a composed prim shall not be allowed to have a composed `typeName` value that refers to the name of an abstract typed schema. Schema types not defined as explicitly abstract shall be treated as concrete.

A typed schema may optionally *inherit* from *one* base schema. This results in the final type of the composed prim containing all the properties defined through its inheritance chain.

If a composed prim has a typed schema authored in the `typeName` field, queries about that prim's type should return whether it is of that type or not, and should include the inheritance chain when making the determination. Although a composed prim is not permitted to be defined with an abstract typed schema, introspection of that prim should still allow determination whether a prim "is of" that type or not (i.e., defined with a concrete typed schema that inherits from an abstract typed schema). That is, a prim is of type T if the prim's `typeName` field value is equal to T or the name of any concrete or abstract ancestor of T.

Consider the following generic schema definition in json:

```
{
  "Foo": {
    "isTyped": true,
    "isAbstract": true,
    "properties": [
      {
        "name": "fooprop",
        "typeName": "int",
        "default": 0
      }
    ]
  },
  "Bar": {
    "isTyped": true,
    "inherits": "Foo",
    "properties": [
      {
        "name": "barprop",
        "typeName": "float",
        "default": 1.0
      }
    ]
  },
  "Baz": {
    "isTyped": true,
    "inherits": "Foo"
```

```

    }
}

```

And a sample scene description using that schema definition:

```

#usda 1.0

def Bar "bar1" {
    string bar1prop = ""
}

def Baz "baz1" {
    vector2f baz1prop = (0.0, 0.0)
}

```

After composition of this simple stage, the prim `bar1` would have an authored `typeName` of `Bar` with three properties defined for it at the paths `/bar1.bar1prop`, `/bar1.barprop`, and `/bar1.foo1prop`. The prim `baz1` would have an authored `typeName` of `Baz` with two properties defined for it at the paths `/baz1.baz1prop` and `baz1.foo1prop`. If these prims were introspected, the following conclusions could be drawn:

- `baz1` is of type `Baz`
- `baz1` is of type `Foo`
- `baz1` is not of type `Bar`
- `bar1` is of type `Bar`
- `bar1` is of type `Foo`
- `bar1` is not of type `Baz`

If the strongest opinion on the `typeName` field for a composed prim is either an abstract schema name or the name of a schema that cannot be identified by the system as a valid schema type name, then the prim is typeless.

Note that the above generic schema definition contained fallback values for its property definitions. As described in [value resolution](#), these values participate in value resolution and will be returned if no authored value for the property exists. This is important in that schema definitions participate in the value resolution of scene description - any composed prim defined with a schema type that cannot be identified by the system cannot have its scene description properly interpreted due to the missing fallback values.

13.3.2 Applied Schemas

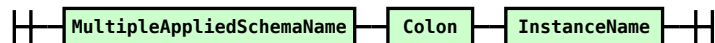
An *applied schema* is similar to a typed schema in that it defines a set of properties, but unlike typed schemas they don't imbue the composed prim with a type. Applied schemas define the properties required to realize an aspect or feature separate from a prim's concrete type. Prim specs may have multiple applied schemas *applied* to them, allowing a composed prim to exhibit multiple different aspects that contribute semantic meaning in different processing scenarios. Applied schemas are applied by authoring the `apiSchemas` metadata field. Like any other metadata field, the values for each contributing opinion are com-

posed into the final value for the composed prim. Since `apiSchemas` is a list op based field, its final value is determined by standard list op resolution [rules](#). Introspection of a composed prim with one or more applied schemas should be able to determine if that prim “has an” applied schema of a specific type applied to it.

An applied schema can be defined as *single* or *multiple*. A single applied schema may only be applied to a composed prim once. A multiple applied schema may be applied to a composed prim many times, and each application is associated with a unique *instance name*, which forms a key part of the property namespace. Whereas a composed prim receives the properties of a single applied schema directly, a composed prim receives the properties of a multiple applied schema through a process of forming the property name by taking the instance name with which it was applied and placing it in a placeholder defined in the schema definition. It is implementation dependent how to define this placeholder. An instance of a multiple applied schema is recorded in scene description in the following form:

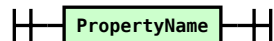
```
Colon <- ':'
```

```
RecordedMultipleAppliedSchemaName <- MultipleAppliedSchemaName Colon InstanceName
```

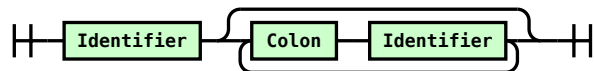


`InstanceName` must follow property identifier rules specified [here](#). That is:

```
InstanceName <- PropertyName
```



```
PropertyName <- Identifier (Colon Identifier)*
```



Note that according to the above grammar definition, instance names themselves may contain multiple namespaces (e.g., identifiers separated by `:`). Consider the following generic schema definition in json, where `<instance_name>` defines the placeholder for where the instance name will be applied to result in the fully resolved property name:

```
{
  "FooBar": {
    "isTyped": false,
    "isMultiple": false,
    "properties": [
      {
        "name": "<instance_name>foo",

```

```

        "typeName": "half",
        "default": 0.5
    }
]
},
"BarBaz": {
    "isTyped": false,
    "isMultiple": true,
    "properties": [
        {
            "name": "<instance_name>:barbazprop",
            "typeName": "string",
            "default": ""
        }
    ]
}
}

```

And a simple scene description using that definition:

```

#usda 1.0

def "foo" (
    apiSchemas = ["FooBar"]
){
    vector3d pt
}

def "bar" (
    apiSchemas = ["FooBar", "BarBaz:my_instance"]
)
{
    string anotherString = "default"
}

```

After composing this simple stage, the prim `foo` would have an authored `apiSchemas` value of `["FooBar"]` indicating that a single applied schema has been applied to the prim at path `/foo`. The resulting prim would have two properties defined for it at the paths `/foo.pt` and `/foo.foobarprop`. The prim `bar` would have an authored `apiSchemas` value of `["FooBar", "BarBaz:my_instance"]`, indicating that two schemas have been applied to the prim, one single applied schema and one multiple applied schema with an instance name of `my_instance`. The resulting prim would have three properties defined for it at the paths `/bar.anotherString`, `/bar.foobarprop`, and `/bar.my_instance:barbazprop`. If these prims were introspected, the following conclusions could be drawn:

- `foo` has a `FooBar` applied to it
- `foo` does not have any instances of `BarBaz` applied to
- `bar` has a `FooBar` applied to it

- bar has one instance of BarBaz applied to it, with an instance name of my_instance

13.3.2.1 Schema Inclusions

Any typed or applied schema may introduce a set of *inclusions*, or rules that specify:

- A set of *built-ins*: additional applied schemas that should be applied when the defining schema is associated with a prim in the context of a composed stage. This set must only be applied schemas.
- A set of *auto-applies*: a set of typed or applied schemas that the defining schema should be applied to, should the prim in the context of a composed stage be of that type or have any of those applied schemas applied to it

The set of built-ins may include multiple applied schemas, with the following restrictions:

- If a typed or single applied schema contains a multiple applied schema in their built-in set, these multiple applied schemas must be named instances (e.g., `MyMultipleAppliedSchema::instanceName` vs. `MyMultipleAppliedSchema`).
- Multiple applied schemas may refer to another multiple applied schema by type or by named instance in their built-in set.
 - If the multiple applied schema is referred to by type, the built-in is applied using the same instance name as the defining multiple applied schema is given when applied.
 - If the multiple applied schema is referred to with an instance name, the built-in is applied by prefixing the instance name used to apply the defining multiple applied schema to the referred instance name of the target multiple applied schema.

The set of auto-applies may include multiple applied schemas, but if present, must be supplied as named instances.

Consider an additional multiple applied schema to the examples above:

```
{
  "FooBar": {
    "isTyped": false,
    "isMultiple": false,
    "properties": [
      {
        "name": "foobarprop",
        "typeName": "half",
        "default": 0.5
      }
    ],
    "inclusions": {
      "auto_applies": ["BazFoo:bazfooinstance"]
    }
  },
  "BarBaz": {
    "isTyped": false,
```

```

        "isMultiple": true,
        "properties": [
            {
                "name": "<instance_name>:barbazprop",
                "typeName": "string",
                "default": ""
            }
        ]
    },
    "BazFoo": {
        "isTyped": false,
        "isMultiple": true,
        "properties": [
            {
                "name": "bazfooprop:<instance_name>",
                "typeName": "int",
                "default": 1
            }
        ],
        "inclusions": {
            "built_ins": ["BarBaz"]
        }
    }
}

```

If FooBar is applied to a prim “Foo” as in the following sample scene description:

```

#usda 1.0

def "foo" (
    apiSchemas = ["FooBar"]
) {
}

```

then the composed prim at the path /foo will have the following applied schemas:

- FooBar
- BazFoo:bazfooinstance
- BarBaz:bazfooinstance

13.3.2.2 Override Properties

Any schema may have an opinion about the value of a property, and the schema with the strongest opinion shall end up providing that value. The ordering of schema opinions is dictated by the **order** on the composed prim. However, there are cases when a schema includes another as a built-in where it needs to override the value of a property, e.g., to include a more reasonable fallback value. Any schema may declare zero or more override properties with the following restrictions:

- The declaration of the property must have the same type of the property it overrides
- The declaration of the property may have a different variability, but it must be ignored
- The declaration of the property must override the same property defined in both directly and indirectly included built-ins

Note that a schema declaring an override property doesn't *define* the property, instead it is treated like an *over* to the property defined by some other schema.

13.3.2.3 The Prim Definition

A prim spec can have both a typed schema and many applied schemas authored onto it, and these schemas may come with additional inclusions, whether they be built-ins or auto-applies. The final set of schemas present on the prim when composed make up the *prim definition*, and as such, determine the final set of properties present on the prim as well as the ordering in which to resolve the fallback value of the property if no stronger authored value is present.

Each schema has an associated prim definition. The prim definition is built for a schema using the following algorithm:

Inputs: Name of the schema

Outputs: Prim Definition for the schema

```
build_prim_definition(schema_name) -> PrimDefinition:
    # create the prim definition object
    # and initialize its properties with that of the defining schema
    prim_definition = create_prim_definition_object()
    prim_definition.schemas.append(schema_name)
    prim_definition.properties = get_properties_from_schema_type(schema_name)
    override_properties = get_override_properties_from_schema_type(schema_name)

    # get the direct schema inclusions
    built_ins = get_built_in_inclusion_list(schema_name)
    auto_applies = get_auto_applies_inclusion_list(schema_name)
    schema_inclusions = built_ins + auto_applies
    prim_definition.schemas += schema_inclusions

    # for each schema inclusion in the order above
    # build its prim definition (if not already built)
    # then compose that prim definition into this one as a weaker prim definition
    for applied_schema in schema_inclusions:
        weaker_prim_definition = build_prim_definition(applied_schema)
        compose_prim_definition(prim_definition, weaker_prim_definition)

    # now compose in each override property value into this prim definition
    # except the variability metadata field, which cannot be overridden
    for override_property in override_properties:
        for metadata_field in override_property.metadata_fields:
```

```

        if !is_variability_metadata_field(metadata_field):
            prim_definition.properties[override_property].set_field(metadata_field,
                override_property.get_field(metadata_field))

    return prim_definition

compose_prim_definition(stronger_prim_definition, weaker_prim_definition):
    # append all schemas
    stronger_prim_definition.schemas += weaker_prim_definition.schemas

    # for each property in the weaker prim definition, determine if the
    # stronger prim definition has it, if not, add it to the stronger
    # prim definition. If the strong prim definition does have it, and
    # the property, check the value of the `default` and `hidden` metadata fields
    # if the stronger property definition has a value for it, skip, otherwise
    # set the value in the stronger property definition
    for prop in weaker_prim_definition.properties:
        if !stronger_prim_definition.has_property(prop):
            stronger_prim_definition.propeerties.add(prop)

    metadata_fields = ["default", "hidden"]
    for metadata_field in metadata_fields:
        if stronger_prim_definition.properties[prop].has_field(metadata_field):
            continue

    stronger_prim_definition.properties[prop].set_field(metadata_field,
        weaker_prim_definition.properties[prop].get_field(metadata_field))

```

The final prim definition for a composed prim is constructed as follows:

- Construct a prim definition for the composed prim with properties determined by composition
- Compose the schema prim definition as a weaker prim definition onto that prim definition

The final prim definition for a composed prim determines the final set of properties that are included for that prim. Note that the final order of schemas for that prim is important, as it will be used to resolve the fallback values of properties for those that do not have authored values. See fallback value resolution for further details.

For instance, consider the prim spec in the following scene description, using the schema definitions above:

```

#usda 1.0

def Bar "myFoo" (
    apiSchemas = ["FooBar"]
){
    double myFoo:prop = 5.0

```

```
}
```

In this case, after composition the prim at path /myFoo would have four properties defined for it:

- /myFoo.myFoo:prop
- /myFoo.barprop
- /myFoo.fooprop
- /myFoo.foobarprop

and introspection of the prim would yield the following:

- myFoo is of type Bar
- myFoo is of type Foo
- myFoo has a FooBar applied to it

Now consider adding an inclusion to the FooBar definition above:

```
{
  "FooBar": {
    "isTyped": false,
    "isMultiple": false,
    "properties": [
      {
        "name": "foobarprop",
        "typeName": "half",
        "default": 0.5
      }
    ],
    "inclusions": {
      "built_ins": ["BarBaz:myInstance"]
    }
  }
}
```

Using the same sample scene description, the prim at path /myFoo would then have five properties defined for it:

- /myFoo.myFoo:prop
- /myFoo.barprop
- /myFoo.fooprop
- /myFoo.foobarprop
- /myFoo.myInstance:barbazprop

and introspection of the prim would yield the following:

- myFoo is of type Bar
- myFoo is of type Foo
- myFoo has a FooBar applied to it
- myFoo has a BarBaz instance applied to it with the instance name myInstance

13.3.2.4 Fallback Value Resolution for Attributes

The prim definition for a composed prim defines how to evaluate the fallback value of an attribute. If a fallback value has to be evaluated, it shall be evaluated in the following order:

- The value provided by the type of the prim (i.e., the typed schema associated with the prim)
- For each applied schema in the ordered set of applied schemas as determined above:
 - The value provided by the applied schema
- The fallback value of the default metadata field (the **unauthorable empty sentinel**)

Note the above is the order in which the prim definition was composed; later schemas in the list were composed as weaker opinions such that the type of the prim always provides the strongest fallback value followed by the applied schemas in the order they were composed in.

In the above example, the schema ordering for the prim definition would be

```
[Bar, FooBar, BarBaz:myInstance]
```

and the schemas would be visited in that order to retrieve a fallback value for an unauthored property.

If instead FooBar had defined an `auto_applies` inclusion as well, for example:

```
{
  "FooBar": {
    "isTyped": false,
    "isMultiple": false,
    "properties": [
      {
        "name": "foobarprop",
        "typeName": "half",
        "default": 0.5
      }
    ],
    "inclusions": {
      "built_ins": ["BarBaz:myInstance"],
      "auto_applies": ["BazFoo:otherInstance"]
    }
  }
}
```

then the schema ordering for the prim definition would be [Bar, FooBar, BarBaz:myInstance, BazFoo:otherInstance].

13.4 Core Schema Types

A number of schemas are introduced as part of this specification to extend the core schema data model. These schemas include those for **color** and **collections**.

14 Color

In order to ensure consistent and accurate colors, color spaces may be used and defined.

Color space information may be specified for:

1. Scene-referred colors.

The *source color space* may be specified for any authored color attribute or asset path identifying a texture or other external color-containing asset.

2. Rendering/lighting calculations involving colors.

These colors are part of Render Settings and not specified here.

14.1 Supported Color Spaces

The canonical color spaces (and their OpenUSD tokens at the time of writing, for reference) are:

14.1.1 Color Space Table

Color Space	OpenUSD Token	Specification or Other References
ACEScsg	lin_ap1_scene	A wide-gamut color space used in the Academy Color Encoding System. See here
ACES2065-1	lin_ap0_scene	SMPTE ST 2065-1.2021
Linear Rec.709 (sRGB)	lin_rec709_scene	See primaries described in: ITU-R BT.709
Linear P3-D65	lin_p3d65_scene	See primaries standardized as the minimum gamut of the Digital Camera Initiatives reference projector standard SMPTE RP 431-2
Linear Rec.2020	lin_rec2020_scene	See primaries described in: ITU-R BT.2020
Linear Adobe RGB	lin_adobergb_scene	N/A
CIE XYZ-D65	lin_ciexyzd65_scene	See CIE 015:2018 - Colorimetry, 4th edition and SMPTE ST 2065-1
sRGB Encoded Rec.709 (sRGB)	srgb_rec709_scene	Colour Science: sRGB EOTF
Gamma 2.2 Encoded Rec.709	g22_rec709_scene	See primaries described in: ITU-R BT.709
Gamma Encoded 1.8 Rec.709	g18_rec709_scene	See primaries described in: ITU-R BT.709

Color Space	OpenUSD Token	Specification or Other References
sRGB Encoded AP1	srgb_ap1_scene	See primaries described in: ACEScg Specification
Gamma Encoded 2.2 AP1	g22_ap1_scene	See primaries described in: ACEScg Specification
sRGB Encoded P3-D65	srgb_p3d65_scene	This is a scene-referred version of Apple's Display P3 color space, and is the same as Linear P3-D65, but sRGB encoded
Gamma 2.2 Encoded AdobeRGB	g22_adobergb_scene	See ICC profile here: Adobe RGB
Data	data	This designation indicates that the asset it describes is actually not color data (e.g. an image file format being used to represent a normal or volume map, etc.), and that no color conversion of the data should be performed.
Unknown	unknown	This designation indicates that the color space is not known (e.g. an image file was created in a tool that did not save the color space information, or provided incorrect color space information).

CIE XYZ-D65 bears some additional explanation. The D65 component in the name is meant to indicate that values transformed to this color space should be adapted to the D65 white point.

For backwards compatibility with older assets, OpenUSD also provides "Raw" (raw) and "Identity" (identity) color space designations that are equivalent to "Data" and "Unknown" designations.

See [Color Space Encodings for Texture Assets and CG Rendering](#) for additional details on these color spaces.

14.2 Core Metadata Extensions

An additional metadata field is added as an extension to the core schema definitions. This field affects [value resolution](#).

14.2.1 **colorSpace: token**

Defines the color space applicable to the spec for which the value is assigned. See [Color Space Inheritance and Resolution](#) for further discussion on how this metadata value affects color space value resolution.

Applies to: Attribute Specs

Fallback Value: ""

14.3 **ColorSpaceDefinitionAPI**

ColorSpaceDefinitionAPI is an API schema for defining a custom color space. Custom color spaces become available for use on prims or for assignment to attributes via the [colorSpace:name](#) property on prims that have applied UsdColorSpaceAPI. Since color spaces [inherit hierarchically](#), a custom color space defined on a prim will be available to all descendants of that prim, unless overridden by a more local color space definition bearing the same name. Locally redefining color spaces within the same layer could be confusing, so that practice is discouraged.

14.3.1 **Attributes**

14.3.1.1 **name : token**

Defines the name of the custom color space.

Variability: uniform

Fallback Value: "custom"

14.3.1.2 **redChroma : float2**

Defines the red chromaticity coordinate.

Variability: varying

Fallback Value: (1.0, 0.0)

14.3.1.3 **greenChroma: float2**

Defines the green chromaticity coordinate.

Variability: varying

Fallback Value: (0.0, 1.0)

14.3.1.4 **blueChroma: float2**

Defines the blue chromaticity coordinate.

Variability: varying

Fallback Value: (0.0, 0.0)

14.3.1.5 whitePoint: float2

Defines the whitepoint chromaticity coordinate.

Variability: varying

Fallback Value: (0.33333333, 0.33333333)

14.3.1.6 gamma: float

Defines the gamma value of the log section.

Variability: varying

Fallback Value: 1.0

14.3.1.7 linearBias: float

Defines the linear bias of the log section.

Variability: varying

Fallback Value: 0.0

14.4 ColorSpaceAPI

ColorSpaceAPI is an API schema that introduces a colorSpace property for authoring scene referred color space opinions. It also provides a mechanism to determine the applicable color space within a scope through inheritance. Accordingly, this schema may be applied to any prim to introduce a color space at any point in a compositional hierarchy.

14.4.1 Attributes

14.4.1.1 colorSpace:name: token

Describes the color space that applies to the attributes with unauthored color spaces on this prim and its descendants.

Variability: uniform

Fallback Value: ""

14.4.2 Example

To specify the source color space, apply the ColorSpaceAPI schema and set the colorSpace:name attribute to the appropriate color space token. The colorSpace:name attribute is a uniform value applied to a prim.

The following example uses ColorSpaceAPI to set the source color space for a texture asset in a Shader prim.

```
def Shader "usduvtexture1"
(
    prepend apiSchemas = ["ColorSpaceAPI"]
```

```

)
{
    uniform token info:id = "UsdUVTexture"
    asset inputs:file = @./assetTexture.png@

    # Specify the source color space for the texture
    uniform token colorSpace:name = "srgb_p3d65_scene"

    # ... other shader attributes omitted ...
}

```

This will set the source color space for all color attributes and color-containing assets in the prim. If you need to provide finer-grain details with source color spaces for specific attributes, set the colorSpace metadata for the attribute.

```

def Material "NewMaterial"
(
    prepend apiSchemas = ["ColorSpaceAPI"]
)
{
    uniform token colorSpace:name = "srgb_p3d65_scene"

    # diffuseColor needs to specify a different color space
    color3f inputs:diffuseColor = (0.2, 0.5, 0.8) (
        colorSpace = "srgb_rec709_scene"
    )
    # ...
}

```

14.4.3 Color Space Inheritance and Resolution

Color spaces can be specified at any level in the scene hierarchy and are inherited by child prims. This hierarchical approach allows for:

- Setting a global color space at the root.
- Providing a color space for a subgraph of the scene.
- Specifying a color space for attributes on individual prims.
- Authoring the color on individual attributes.

The color space that should be applied to an attribute has special value resolution semantics, and the resolution order shall be as follows:

1. Check if the attribute has an explicitly assigned color space.
2. Check if the attribute's prim has the ColorSpaceAPI schema applied.
3. Search up the hierarchy until a prim with ColorSpaceAPI is found.
4. If no color space is found, an empty token is returned. In this case, the color space must be assumed to be the **default color space**.

The following example illustrates how color space information is propagated and resolved.

The “Root” prim has the ColorSpaceAPI schema applied and specifies a color space of lin_rec709_scene. All child prims of “Root” will inherit lin_rec709_scene as their color space unless explicitly overridden. Three child prims are defined, with color space resolution as follows:

- The “Material1” child prim overrides the color space from its parent. All attributes of “Material1” will have srgb_p3d65_scene as their color space.
- The “Material2” child prim does not provide an opinion for colorSpace:name and therefore will inherit the lin_rec709_scene color space from “Root”. Note that because “Material2” also does not have the ColorSpaceAPI schema applied, if it did provide an opinion for colorSpace:name, this would *not* be used in color space resolution.
- The “Material3” child prim also does not provide an opinion for colorSpace:name and therefore will inherit the lin_rec709_scene color space from “Root”. However, the inputs:diffuseColor attribute *does* explicitly define a color space of srgb_rec709_scene, which will be used as the source color space for that attribute only.

```
def Xform "Root" (
  prepend apiSchemas = ["ColorSpaceAPI"]
)
{
  uniform token colorSpace:name = "lin_rec709_scene"

  def Material "Material1"
  (
    prepend apiSchemas = ["ColorSpaceAPI"]
  )
  {
    # Material1 overrides Root's color space with srgb_p3d65_scene
    uniform token colorSpace:name = "srgb_p3d65_scene"

    color3f inputs:diffuseColor = (0.2, 0.5, 0.8)
  }

  def Material "Material2"
  {
    # Material2 inherits Root's lin_rec709_scene color space
    color3f inputs:diffuseColor = (0.2, 0.5, 0.8)
  }

  def Material "Material3"
  {
    # Even though Material3 inherits lin_rec709_scene from Root,
    # the diffuseColor attribute specifically uses srgb_rec709_scene
    color3f inputs:diffuseColor = (0.2, 0.5, 0.8) (
      colorSpace = "srgb_rec709_scene"
    )
  }
}
```

}

14.4.4 Default Color Space

The default color space is Linear Rec.709. This color space is used for the source color space for prims and attributes when color space resolution takes place and no authored color space is found.

14.4.5 Glossary of Color Terms

Chromaticity: The quality of color regardless of luminance.

Color gamut: The range of colors that can be represented or displayed.

Color space: A specific organization of colors that provides a common reference.

Display-referred: Color values adapted for direct display on a particular device.

Gamma: A non-linear operation applied to encode or decode luminance values.

Linear color space: A color space where values are directly proportional to light energy.

OCIO (OpenColorIO): An open-source color management solution used in visual effects and animation.

Primaries: The fundamental red, green, and blue components that define a color space.

Scene-referred: Color values that represent actual light in a scene before display adaptation.

Transfer function: The mathematical relationship between numeric color values and displayed brightness.

White point: The reference neutral point in a color space, typically defined as a specific color temperature.

15 Collections

Collections are an **applied schema** that combine relationships with expansion rules to identify its member scene objects. Relationships specify targets via composed path list operations. Collections expand this concept by using a pair of relationships, one to define a set of paths to *include* in the collection and one to define a set of paths to *exclude* from the collection. This mechanism can be used to include a group of prims from the scene while excluding certain children from that included group.

Collections are introduced onto a prim via the CollectionAPI multiple applied schema, allowing a prim to contain multiple collections with different instance names referring to different sets of included / excluded objects.

15.1 CollectionAPI

The CollectionAPI multiple applied schema introduces the ability to define a pair of included / excluded relationships to objects in a scene.

15.1.1 Attributes

15.1.1.1 **expansionRule:** token

Describes how paths that are included in the collection are evaluated to determine the collection's members.

Variability: uniform

Allowed Values: "explicitOnly", "expandPrims", "expandPrimsAndProperties"

Fallback Value: "expandPrims"

15.1.1.2 **includeRoot:** bool

Specifies if the collection should include the absolute root path in the collection's included target paths. A separate attribute is required because relationships **cannot explicitly target** the absolute root path.

Variability: uniform

Fallback Value: false

15.1.1.3 **includes:** rel[]

Defines the list of relationship targets that will be included in this collection. These relationship targets can include prims and properties.

Fallback Value: []

15.1.1.4 **excludes:** rel[]

Defines the list of relationship targets that will be excluded from the collection based on the context of the included group. These relationship targets can include prims and properties,

but may not target a property referring to another collection.

Fallback Value: []

15.1.1.5 collection:<instance_name>: opaque

Defines the property that can be referenced to allow a collection to include another collection. If a relationship target to a collection is added to the includes group, the runtime must consider the includes list of that collection in its own set of rules to determine collection membership (recursively).

15.2 Authoring and Evaluating Collections

Authoring collection instances on a prim is done by applying the CollectionAPI multiple applied schema with an instance name and authoring the includes and excludes relationship targets (and the expansionRule and includeRoot attributes, if applicable). Consider the following scene description:

```
over "foo" {
  over "bar" (
    prepend apiSchemas = "CollectionAPI:rooms"
  )
  {
    rel collection:rooms:includes = [
      </foo/bar/office>
    ]
  }
}
```

In this scene description, an instance rooms of the CollectionAPI schema has been applied to the prim at path /foo/bar. The includes targets includes a child prim of /foo/bar (office). Since the fallback value for expansionRule is expandPrims (see more below), all child prims of /foo/bar/office will be included in the collection instance.

The author may exclude certain child prims using the excludes rule on the collection instance. For example:

```
over "foo" {
  over "bar" (
    prepend apiSchemas = "CollectionAPI:rooms"
  )
  {
    rel collection:rooms:includes = [
      </foo/bar/office>
    ]
    rel collection:rooms:excludes = [
      </foo/bar/office/bookshelf>
    ]
  }
}
```

```

    }
}

```

In the above case, all the child prims of `/foo/bar/office` will be included in the rooms collection instance *except* for `/foo/bar/office/bookshelf` and its descendants, because it is a member of the set of target relationships defined in the `excludes` property.

The value of `expansionRule` determines the final set of paths that are considered included members of the collection instance. Interpreting the semantics of each of the allowed values must be done according to the following rules:

- `expandPrims`: the set of included paths shall consist of the paths of all prims including and descendant to the specified prim path.
- `expandPrimsAndProperties`: the set of included paths shall consist of the paths of all prims and all properties within those prims including and descendant to the specified prim path.
- `explicitOnly`: the set of included paths shall consist *only* of the paths explicitly listed in the `includes` set (and not excluded by the `excludes` set).

Consider the following scene description:

```

def "foo" {
  def "bar" (
    prepend apiSchemas = "CollectionAPI:bar1"
  )
  {
    rel collection:bar1:includes = [
      </foo/bar/baz>
    ]
    rel collection:bar1:excludes = [
      </foo/bar/baz.foobaz>
    ]

    def "baz" {
      uniform token foobar = "foobar"
      uniform token foobaz = "foobaz"

      def "boo" {
      }
    }
  }
}

```

The evaluated included paths for each of the rules would be as follows:

- `expandPrims`: `[/foo/bar/baz, /foo/bar/baz/boo]`
- `expandPrimAndPropertyPaths`: `[/foo/bar/baz, /foo/bar/baz.foobar, /foo/bar/baz/-boo]`
- `explicitOnly`: `[foo/bar/baz]`

Relationship targets for includes can also include property paths referring to other collections. In this case, when evaluated, the set of property paths that would be included in the target collection are also included in targeting collection (recursively, if the targeted collection also targeted another collection, etc.)

Expanding on the scene description above:

```
def "foo" {
  def "bar" (
    prepend apiSchemas = "CollectionAPI:bar1"
  )
  {
    rel collection:bar1:includes = [
      </foo/bar/baz>,
      </foo/fooboo.collection:foobool>
    ]
    rel collection:bar1:excludes = [
      </foo/bar/baz.foobaz>
    ]

    def "baz" {
      uniform token foobar = "foobar"
      uniform token foobaz = "foobaz"

      def "boo" {
      }
    }
  }

  def "fooboo" (
    prepend apiSchemas = "CollectionAPI:foobool"
  )
  {
    rel collection:foobool:includes = [
      </foo/fooboo/fooboobar>
    ]
    def "fooboobar" {
    }
  }
}
```

Using the `expandPrims` value for the `expansionRule`, the `bar1` collection instance would include:

```
[/foo/bar/baz, /foo/bar/baz/boo, /foo/fooboo/fooboobar]
```

Note that the set of target relationships in `excludes` should not contain *orphaned* excluded paths. That is, excluded paths must refer to a subset of those included in the included paths. If orphaned excluded paths are present, their presence is treated as inert.

16 Core File Formats

This section will outline the three core file formats that are required to be implemented for USD.

16.1 Compatibility with Document Versions

All readers, regardless of format, must be able to read all previous versions within the same MAJOR version, unless:

- The specification provides a minimum version of a document to support
- Extenuating security issues are present within a document version

For example, a hypothetical version 1.2.1 compatible parser, must read all 1.0, 1.1, 1.2 version documents up to the current supported version.

It is not required to read documents that have newer MAJOR or MINOR versions than the current parser, but all future PATCH versions must be readable.

However, if a particular PATCH version is known to carry a security risk or other breaking risks, files bearing that version may be rejected by a later parser. An implementation should consider providing a method to migrate a file bearing a rejected version to the latest version.

When reading unsupported document versions, it is recommended to surface an error upon discovering the document version is unsupported. Parsers must not simply ignore unsupported elements of the grammar as that may unintentionally change the intention of the document.

16.2 Text

USD represents its scene data in a bespoke text grammar as USDA files. They may be stored with the `.usda` or `.usd` extension.

16.2.1 Encoding

While the USDA name originally implied that the files were ASCII, USDA files today are encoded in the UTF-8 standard, as defined by [ISO/IEC 10646:2020](#).

16.2.2 Grammar Conventions

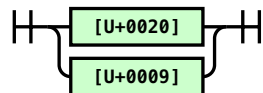
The Specification will describe the parsing grammar for USDA documents, which will hereafter be referred to as “the grammar”, or as “the USDA grammar”. The grammar is specified using PEG notation as defined in [1].

The conventions used here are the same as those described in the [path grammar](#).

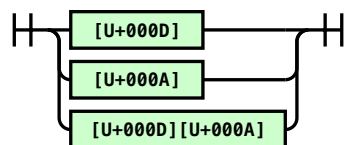
16.2.3 Whitespace Formatting

Simple whitespace denotes basic logical separation rules between two parts of the document. The USDA document format is not indentation sensitive and items on independent lines may be indented at different levels than each other. End of lines in USDA can be either carriage return (CR), line feed (LF), or a combination of CRLF.

```
Space <- [U+0020] /  
        [U+0009]
```



```
CrLf <- [U+000D] /  
        [U+000A] /  
        [U+000D][U+000A]
```



16.2.4 Special Characters

Certain characters are used throughout the grammar for purposes such as denoting e.g, lists, tuples, start and end of blocks, etc. Explicit rules are made for these characters to make it easier to read where these characters are used in more complex rule definitions.

```
ForwardSlash <- '/'
```

```
Backslash <- '\'
```

```
SingleQuote <- "'"
```

```
DoubleQuote <- '"'
```

```
LeftParen <- '('
```

```
RightParen <- ')'
```

```
LeftBracket <- '['
```

```
RightBracket <- ']'
```

```
LeftCurlyBrace <- '{'
```

```

RightCurlyBrace <- '}'

LeftAngleBracket <- '<'

RightAngleBracket <- '>'

Ampersand <- '&'

Asterisk <- '*'

At <- '@'

Colon <- ':'

Comma <- ','

Dot <- '.'

Equals <- '='

Minus <- '-'

Plus <- '+'

Pound <- '#'

SemiColon <- ';'

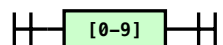
Underscore <- '_'

```

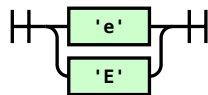
16.2.5 Character and Numeric Literals

Literals represent the basic production rules where literal characters are given semantic meaning. These are production rules that represent characters, numbers, and strings. As USDA must be UTF-8 encoded, valid characters are interpreted as Unicode code points and generally comprise characters that do not exist in the private-use or non-character code point ranges.

```
Digit <- [0-9]
```



```
Exponent <- 'e' / 'E'
```

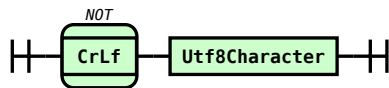


```

Utf8Character <- [U+0020-U+007E] /
                  [U+00A0-U+D7FF] /
                  [U+F900-U+FDCE] /
                  [U+FDFF-U+FEFF] /
                  [U+10000-U+1FFFF] /
                  [U+20000-U+2FFFF] /
                  [U+30000-U+3FFFF] /
                  [U+40000-U+4FFFF] /
                  [U+50000-U+5FFFF] /
                  [U+60000-U+6FFFF] /
                  [U+70000-U+7FFFF] /
                  [U+80000-U+8FFFF] /
                  [U+90000-U+9FFFF] /
                  [U+A0000-U+AFFFF] /
                  [U+B0000-U+BFFFF] /
                  [U+C0000-U+CFFFF] /
                  [U+D0000-U+DFFFF] /
                  [U+E0000-U+EFFFF]
  
```

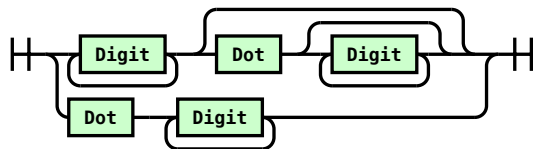
```

NonCrLfUtf8Character <- !CrLf Utf8Character
  
```



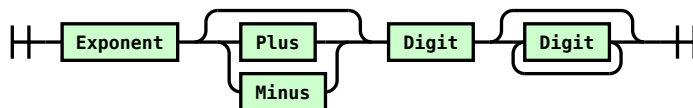
```

BasePart <- (Digit)+ /
            (Digit)+ Dot (Digit)* /
            Dot (Digit)+
  
```



```

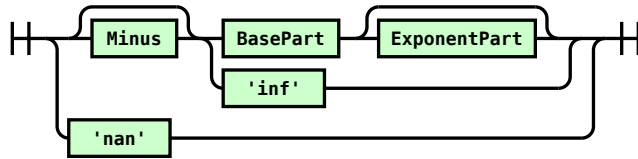
ExponentPart <- Exponent (Plus / Minus)? Digit (Digit)*
  
```



```

Number <- (Minus)? BasePart (ExponentPart)? /
    'inf' /
    Minus 'inf' /
    'nan'

```



```

HexDigit <- [0-9] /
    [A-F]

```

```

OctDigit <- [0-7]

```

```

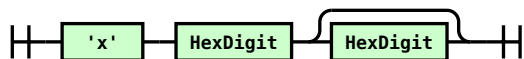
EscapeSingleCharacter <- 'a' /
    'b' /
    'f' /
    'n' /
    'r' /
    't' /
    'v' /
    SingleQuote /
    DoubleQuote

```

```

EscapeHex <- 'x' HexDigit (HexDigit)?

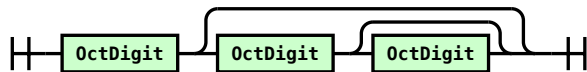
```



```

EscapeOct <- OctDigit (OctDigit (OctDigit)?)?

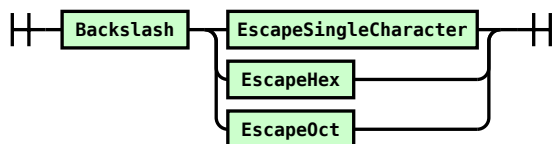
```



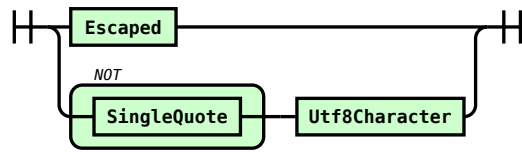
```

Escaped <- Backslash (EscapeSingleCharacter /
    EscapeHex /
    EscapeOct)

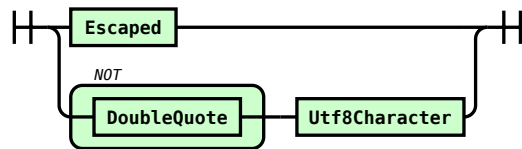
```



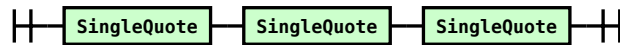
```
MultilineSingleQuoteContents <- Escaped /
                               !(SingleQuote) Utf8Character
```



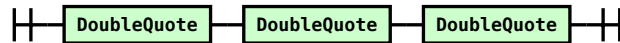
```
MultilineDoubleQuoteContents <- Escaped /
                                !(DoubleQuote) Utf8Character
```



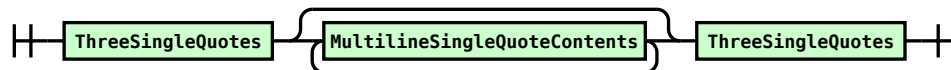
```
ThreeSingleQuotes <- SingleQuote SingleQuote SingleQuote
```



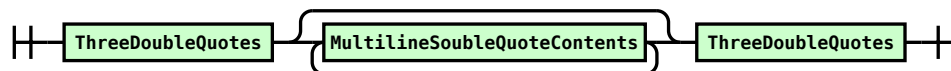
```
ThreeDoubleQuotes <- DoubleQuote DoubleQuote DoubleQuote
```



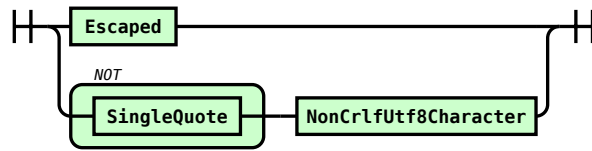
```
MultilineSingleQuoteString <- ThreeSingleQuotes
                              (MultilineSingleQuoteContents)*
                              ThreeSingleQuotes
```



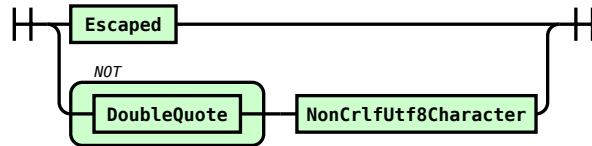
```
MultilineDoubleQuoteString <- ThreeDoubleQuotes
                              (MultilineDoubleQuoteContents)*
                              ThreeDoubleQuotes
```



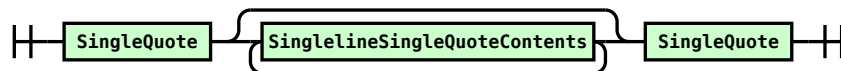
```
SinglelineSingleQuoteContents <- Escaped /
                                !(SingleQuote) NonCrLfUtf8Character
```



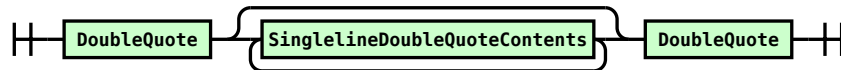
```
SinglelineDoubleQuoteContents <- Escaped /
                                !(DoubleQuote) NonCrLfUtf8Character
```



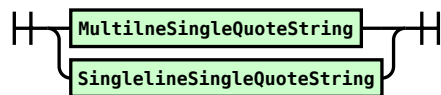
```
SinglelineSingleQuoteString <- SingleQuote (SinglelineSingleQuoteContents)* SingleQuote
```



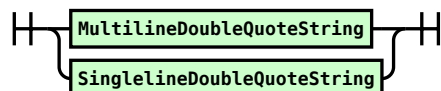
```
SinglelineDoubleQuoteString <- DoubleQuote (SinglelineDoubleQuoteContents)* DoubleQuote
```



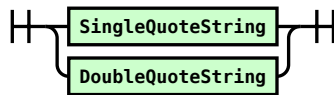
```
SingleQuoteString <- MultilineSingleQuoteString /
                    SinglelineSingleQuoteString
```



```
DoubleQuoteString <- MultilineDoubleQuoteString /
                    SinglelineDoubleQuoteString
```



```
String <- SingleQuoteString /
          DoubleQuoteString
```

16.2.5.1 Double-Precision Floating Point Representation when Writing USDA content

The conversion of a double-precision floating point number to its string representation shall follow these rules:

1. Special Values

- 1.1. If the value is positive infinity, it shall be represented as "inf"
- 1.2. If the value is negative infinity, it shall be represented as "-inf"
- 1.3. If the value is Not-a-Number, it shall be represented as "nan"

2. Finite Values

- 2.1. The conversion shall produce the shortest string that correctly represents the value such that parsing the string returns the original value
- 2.2. The representation shall use either decimal or scientific notation based on the following criteria:
 - a) Scientific notation shall be used when the decimal exponent is less than -6
 - b) Scientific notation shall be used when the decimal exponent is greater than or equal to 15
 - c) Decimal notation shall be used otherwise
- 2.3. When scientific notation is used:
 - a) It shall consist of one significant digit before the decimal point
 - b) It shall use the character 'e' or 'E' to denote the exponent
 - c) The exponent shall be represented as a signed decimal integer
- 2.4. The representation shall not include (even though the grammar allows it):
 - a) Trailing decimal points
 - b) Unnecessary trailing zeros in the fractional part
 - c) Positive signs for positive exponents
 - d) Leading zeros in the integer part except when representing values less than 1

3. Examples of Compliant Representations:

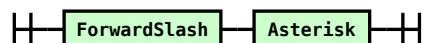
- 0.0000001 -> "0.0000001"
- 0.00000001 -> "1e-7"
- 10000000000000000.0 -> "1e16"
- 1000.0 -> "1000"
- 0.0 -> "0"

16.2.6 Comments, Complex Whitespace, and Separators

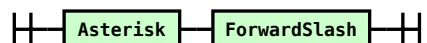
USDA supports the use of comments, which are human-readable annotations that do not contribute to scene description.

USDA supports a varying set of comment representations and these comments contribute to more complex whitespace definitions that are used in production rules throughout the grammar. Separators are used to denote separation between logical items in the grammar, whether they be explicit lists of items (e.g., lists, tuples, etc.) or implicit ones (e.g., prims, properties, etc.).

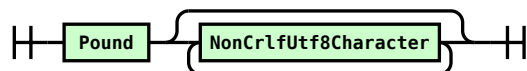
```
CppStyleMultilineOpen <- ForwardSlash Asterisk
```



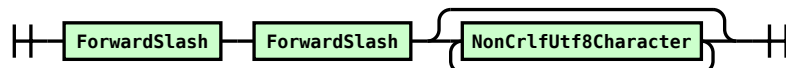
```
CppStyleMultilineClose <- Asterisk ForwardSlash
```



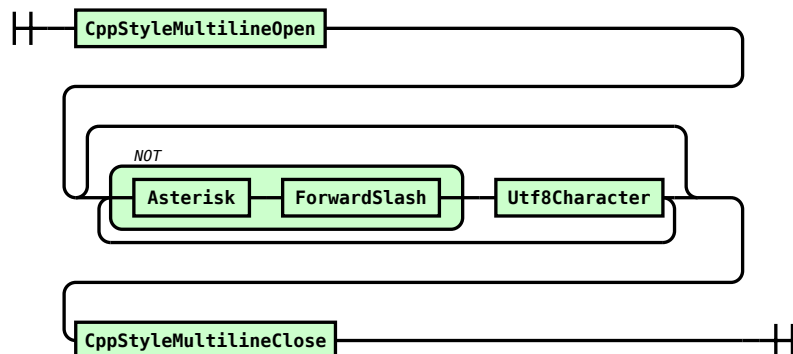
```
PythonStyleComment <- Pound (NonCrLfUtf8Character)*
```



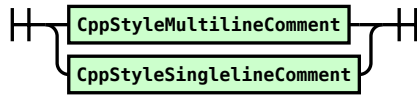
```
CppStyleSinglelineComment <- ForwardSlash ForwardSlash (NonCrLfUtf8Character)*
```



```
CppStyleMultilineComment <- CppStyleMultilineOpen ↵  
                             (! (Asterisk ForwardSlash) Utf8Character)* ↵  
                             CppStyleMultilineClose
```



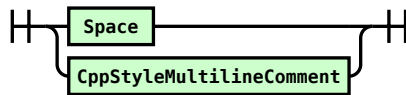
```
CppStyleComment <- CppStyleMultilineComment /
                  CppStyleSinglelineComment
```



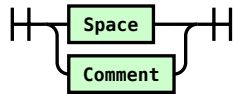
```
Comment <- PythonStyleComment /
          CppStyleComment
```



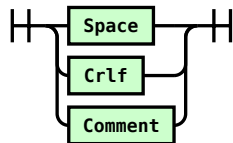
```
InlinePadding <- Space /
                CppStyleMultilineComment
```



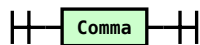
```
SinglelinePadding <- Space /
                   Comment
```



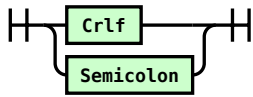
```
MultilinePadding <- Space /
                  Crlf /
                  Comment
```



```
ListSeparator <- Comma
```



```
StatementSeparator <- Crlf /
                    Semicolon
```



16.2.7 Keywords

Keywords are specific sequences of literals reserved by USDA with specific semantic meaning.

Note: In almost all cases, a keyword is identified specifically in the production rule it is a part of. The keywords are listed here specifically to identify them as reserved words for the *KeywordlessIdentifier* production in cases where it is required to separate semantic meaning between any *BaseIdentifier* and an *Identifier* that is not a *Keyword*.

```
Keyword <- 'append' /
          'bezier' /
          'class' /
          'connect' /
          'curve' /
          'custom' /
          'customData' /
          'def' /
          'delete' /
          'dictionary' /
          'doc' /
          'held' /
          'hermite' /
          'inherits' /
          'kind' /
          'linear' /
          'loop' /
          'nameChildren' /
          'None' /
          'none' /
          'offset' /
          'oscillate' /
          'over' /
          'payload' /
          'post' /
          'pre' /
          'prepend' /
          'properties' /
          'references' /
```

```

'relocates' /
'rel' /
'reorder' /
'repeat' /
'reset' /
'rootPrims' /
'scale' /
'sloped' /
'subLayers' /
'specializes' /
'spline' /
'timeSamples' /
'uniform' /
'variantSet' /
'variantSets' /
'variants' /
'varying'

```

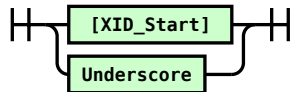
16.2.8 Identifiers

USD layers consist of several semantic rules that contain *identifiers*.

```

XidStart <- [XID_Start] /
            Underscore

```



```

XidContinue <- [XID_Continue]

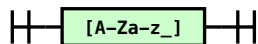
```



```

AsciiStart <- [A-Za-z_]

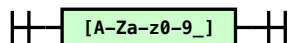
```



```

AsciiContinue <- [A-Za-z0-9_]

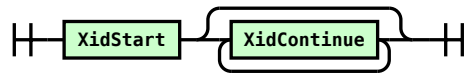
```



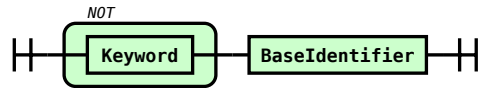
```

BaseIdentifier <- XidStart (XidContinue)*

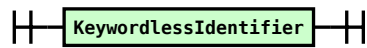
```



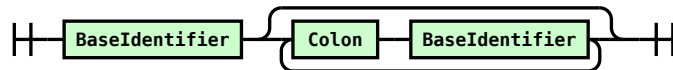
```
KeywordlessIdentifier <- !(Keyword) BaseIdentifier
```



```
Identifier <- KeywordlessIdentifier
```



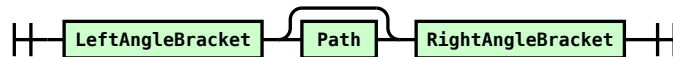
```
NamespacedName <- BaseIdentifier (Colon BaseIdentifier)*
```



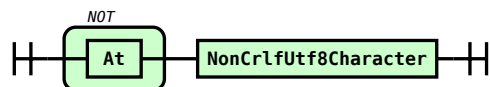
16.2.9 Paths and Asset References

Paths are used in several places in the grammar to define references to prims or properties within a scene. The productions for Path reference the **Path** production [here](#). This ensures interpretation of paths is consistent when parsing both paths and layers. Assets can also be referenced from a layer and scales or offsets within those assets can be used when bringing in content from another layer.

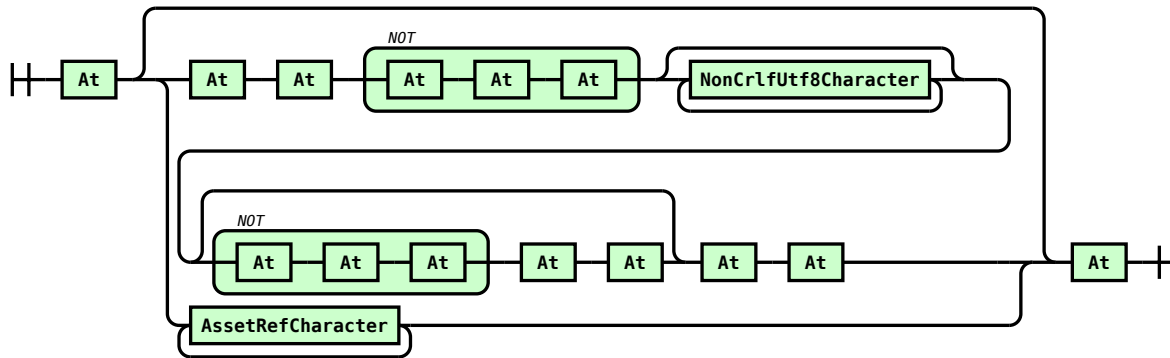
```
PathRef <- LeftAngleBracket Path? RightAngleBracket
```



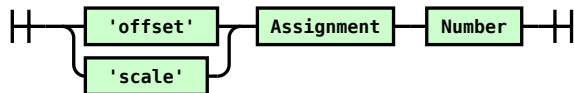
```
AssetRefCharacter <- !(At) NonCrLfUtf8Character
```



```
AssetRef <- At At At (!(At At At)(NonCrLfUtf8Character)*) ↵
                (!(At At At)(At At))? At At At /
                At (AssetRefCharacter)* At
```



```
LayerOffset <- 'offset' Assignment Number /
               'scale' Assignment Number
```



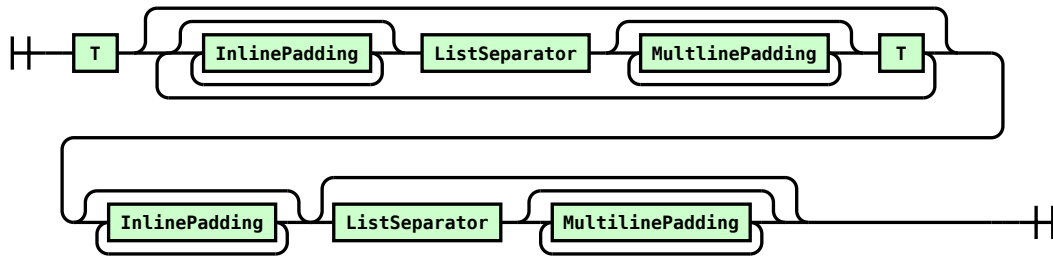
Examples of asset path declarations are given below:

Declaration	Description
@texture.png@	References an external file path called texture.png
@car.usdz[0/texture.png]@	References an asset path at 0/texture.png within the car.usdz
@car.usdz@</models/window>	References a prim at the path /models/window within the car.usdz file
</models/window>	References a prim at the path /models/window within the current USD stage

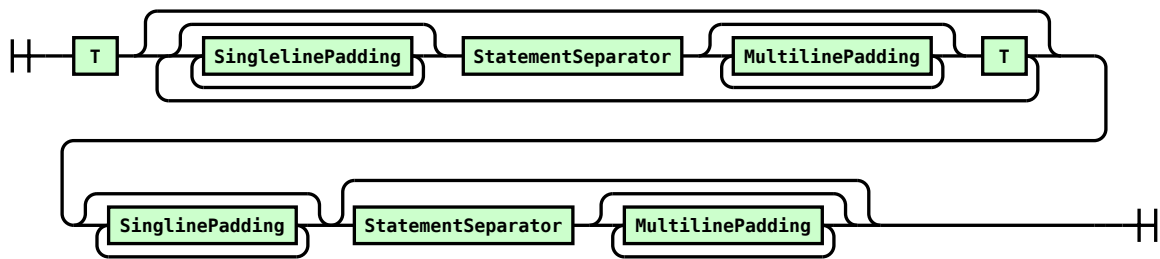
16.2.10 Lists and Statement Contents

Though not strictly PEG grammar rules, this section highlights a generic production that makes it easier to read complex productions that contain lists or sequences of statements. The generic is denoted with the letter T which can be replaced with a real production when seen in the grammar.

```
List<T> <- T ((InlinePadding)* ListSeparator (MultilinePadding)* T)*
           (InlinePadding)* (ListSeparator (MultilinePadding)*)?
```



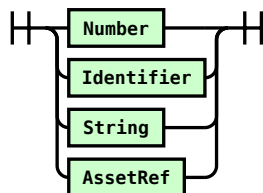
```
Sequence<T> <- T ((SinglelinePadding)* StatementSeparator (MultilinePadding)* T)* ↵
                (SinglelinePadding)* (StatementSeparator (MultilinePadding)*)?
```



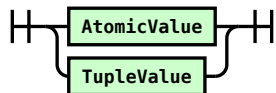
16.2.11 Values

Values are data stored in fields at a particular location (e.g., attribute, reference, etc.). These are typically found on the right hand side of assignment statements and represent simple values (such as numbers, identifiers, strings, and asset references) and complex values (such as tuples, lists, dictionaries, etc.). Typically, complex values store sets of values that could be simple, complex, or heterogeneous. In certain cases, specific productions are called out for e.g., dictionaries that can only have key / value pairs represented as strings, etc.

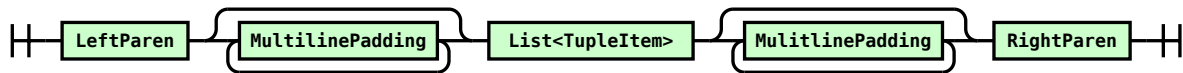
```
AtomicValue <- Number /
               Identifier /
               String /
               AssetRef
```



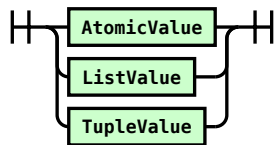
```
TupleItem <- AtomicValue /
             TupleValue
```

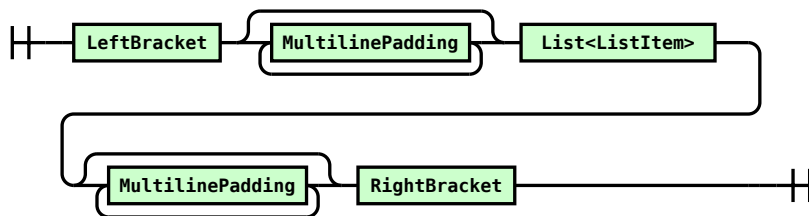
```
TupleValue <- LeftParen (MultilinePadding)* List<TupleItem> (MultilinePadding)* RightParen
```



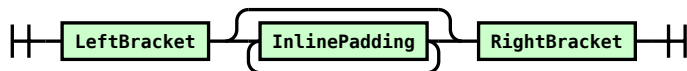
```
ListItem <- AtomicValue /  
           ListValue /  
           TupleValue
```



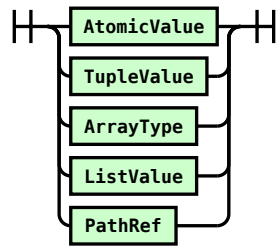
```
ListValue <- LeftBracket (MultilinePadding)* List<ListItem> ↵  
             (MultilinePadding)* RightBracket
```



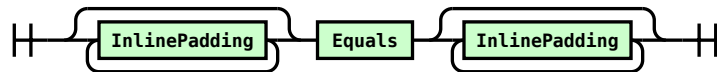
```
ArrayType <- LeftBracket (InlinePadding)* RightBracket
```



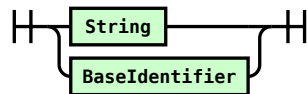
```
TypedValue <- AtomicValue /  
              TupleValue /  
              ArrayType /  
              ListValue /  
              PathRef
```



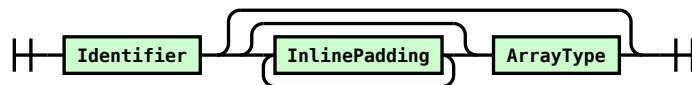
```
Assignment <- (InlinePadding)* Equals (InlinePadding)*
```



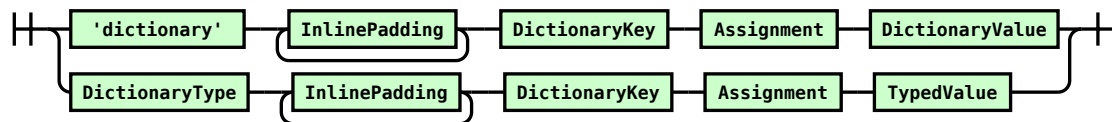
```
DictionaryKey <- String /  
                BaseIdentifier
```



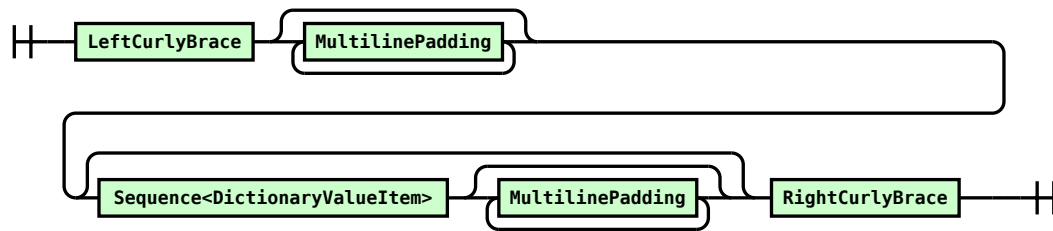
```
DictionaryType <- Identifier ((InlinePadding)* ArrayType)?
```



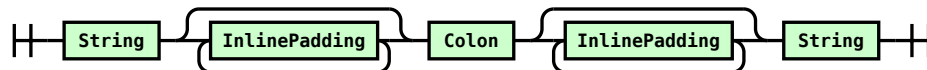
```
DictionaryValueItem <- 'dictionary' (InlinePadding)+ DictionaryKey Assignment DictionaryValue /  
                      DictionaryType (InlinePadding)+ DictionaryKey Assignment TypedValue
```



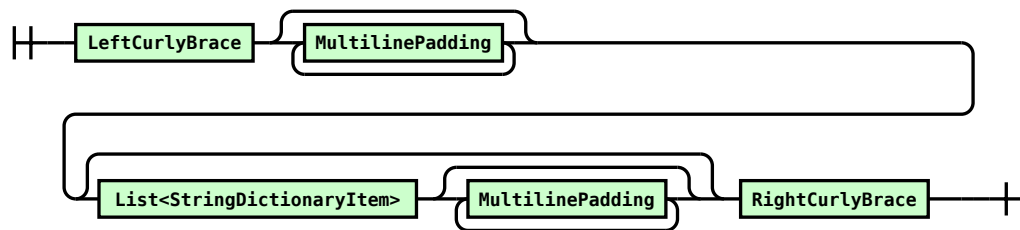
```
DictionaryValue <- LeftCurlyBrace (MultilinePadding)* ↵  
                      (Sequence<DictionaryValueItem> (MultilinePadding)*)? RightCurlyBrace
```



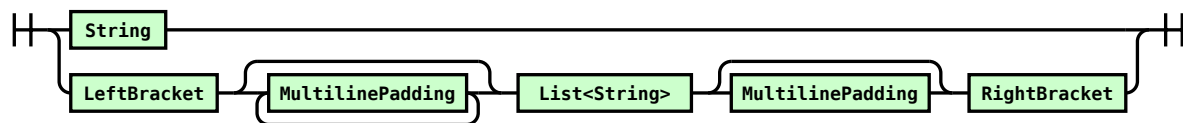
```
StringDictionaryItem <- String (InlinePadding)* Colon (InlinePadding)* String
```



```
StringDictionary <- LeftCurlyBrace (MultilinePadding)* ↵  
                      (List<StringDictionaryItem> (MultilinePadding)*)? RightCurlyBrace
```



```
NameList <- String /  
             LeftBracket (MultilinePadding)* List<String> (MultilinePadding)? RightBracket
```

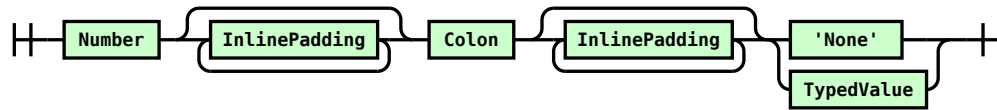


Note: While the grammar itself makes no statement about the type of values in lists and tuples, in practice these values should be homogenous in type for downstream use.

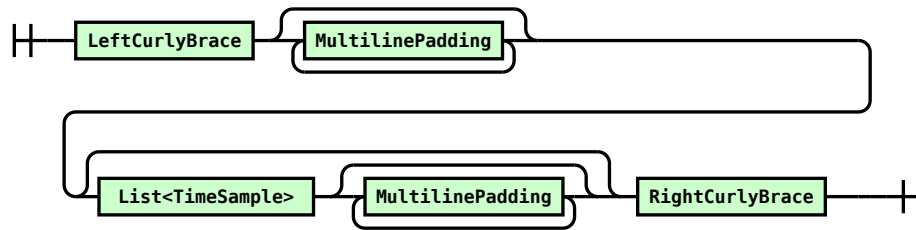
16.2.12 Time Samples

Time samples are used to describe a key-value mapping that maps time code ordinates to values such that values of an attribute can change over time.

```
TimeSample <- Number (InlinePadding)* Colon (InlinePadding)*  
              ('None' / TypedValue)
```



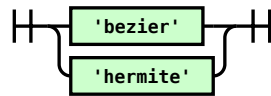
```
TimeSampleMap <- LeftCurlyBrace (MultilinePadding)* ←
                    (List<TimeSample> (MultilinePadding)*)? RightCurlyBrace
```



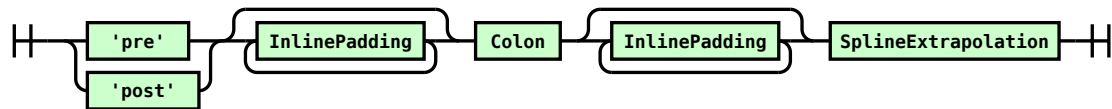
16.2.13 Splines

Splines are used to define spline curves as a first class data type.

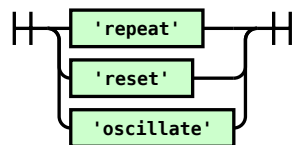
```
SplineCurveTypeItem <- 'bezier' /
                        'hermite'
```



```
SplineExtrapolationItem <- 'pre' (InlinePadding)* Colon (InlinePadding)* SplineExtrapolation /
                            'post' (InlinePadding)* Colon (InlinePadding)* SplineExtrapolation
```



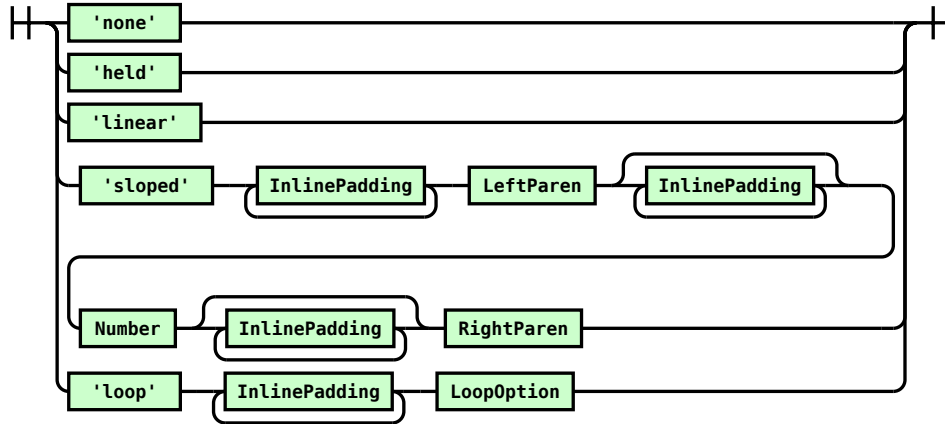
```
LoopOption <- 'repeat' /
              'reset' /
              'oscillate'
```



```

SplineExtrapolation <- 'none' /
                        'held' /
                        'linear' /
                        'sloped' (InlinePadding)+ LeftParen (InlinePadding)* ↵
                        Number (InlinePadding)* RightParen /
                        'loop' (InlinePadding)+ LoopOption

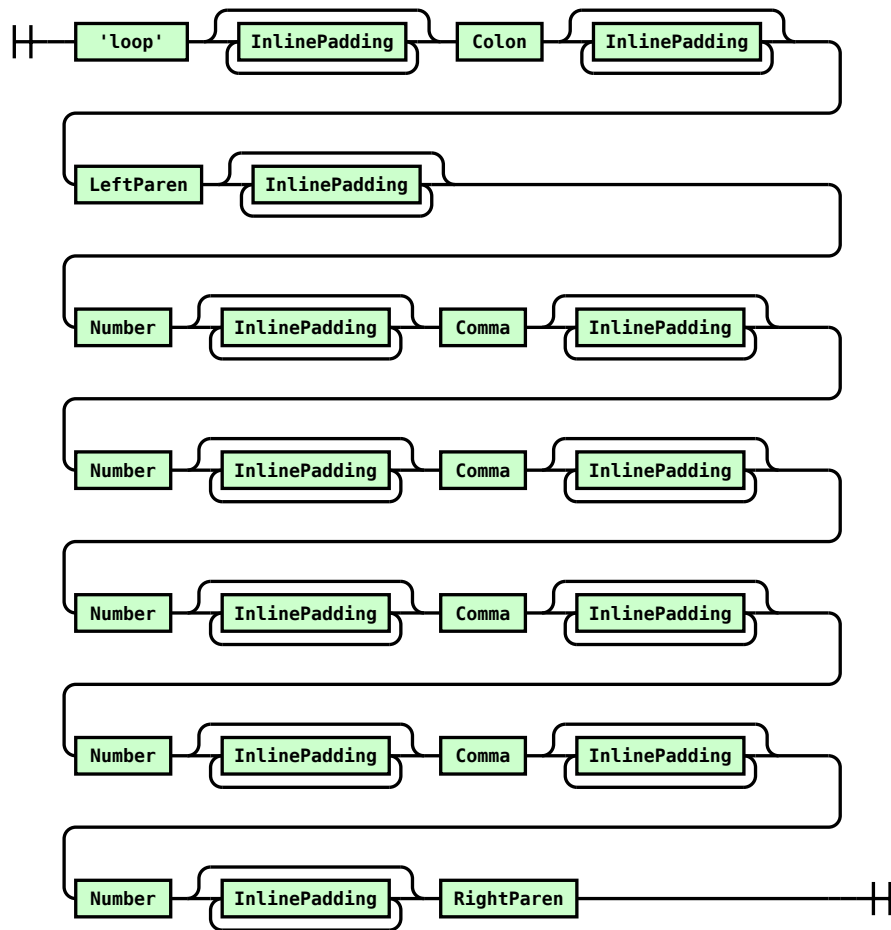
```



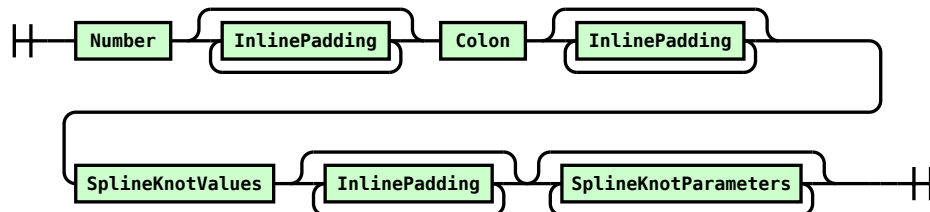
```

SplineLoopItem <- 'loop' (InlinePadding)* Colon (InlinePadding)* ↵
                  LeftParen (InlinePadding)* ↵
                  Number (InlinePadding)* Comma (InlinePadding)* ↵
                  Number (InlinePadding)* Comma (InlinePadding)* ↵
                  Number (InlinePadding)* Comma (InlinePadding)* ↵
                  Number (InlinePadding)* Comma (InlinePadding)* ↵
                  Number (InlinePadding)* RightParen

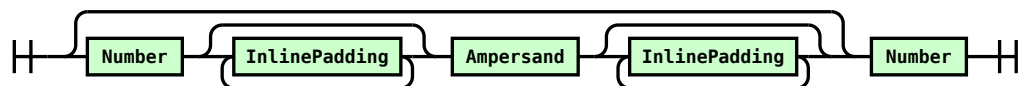
```



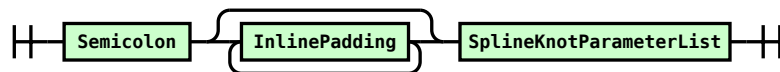
```
SplineKnotItem <- Number (InlinePadding)* Colon (InlinePadding)* ↵
                  SplineKnotValues (InlinePadding)* (SplineKnotParameters)*
```



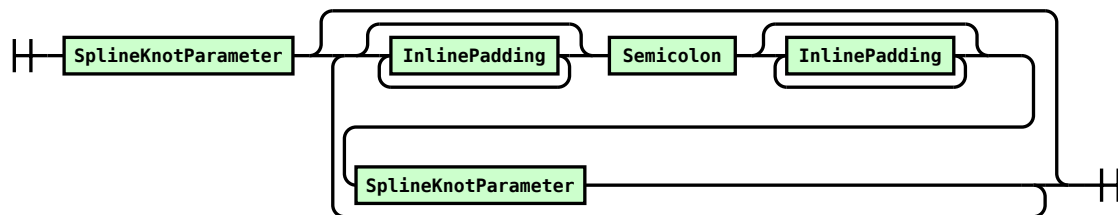
```
SplineKnotValues <- Number (InlinePadding)* Ampersand (InlinePadding)* Number /
                    Number
```



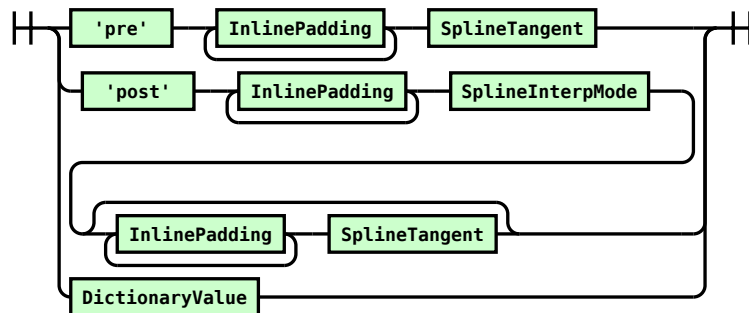
```
SplineKnotParameters <- Semicolon (InlinePadding)* SplineKnotParameterList
```



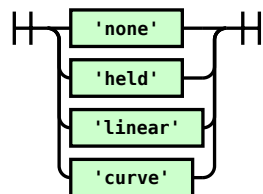
```
SplineKnotParameterList <- SplineKnotParameter
  ((InlinePadding)* Semicolon (InlinePadding)* ↵
  SplineKnotParameter)*
```



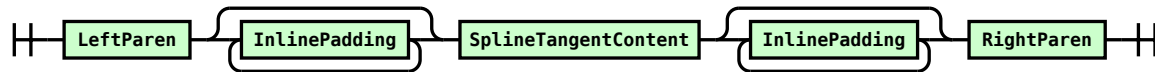
```
SplineKnotParameter <- 'pre' (InlinePadding)+ SplineTangent /
  'post' (InlinePadding)+ SplineInterpMode ↵
  ((InlinePadding)+ SplineTangent)? /
  DictionaryValue
```



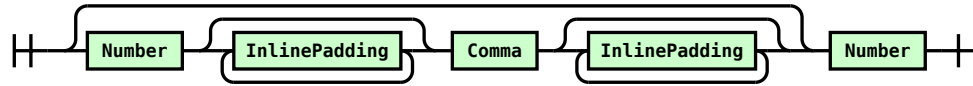
```
SplineInterpMode <- 'none' /
  'held' /
  'linear' /
  'curve'
```



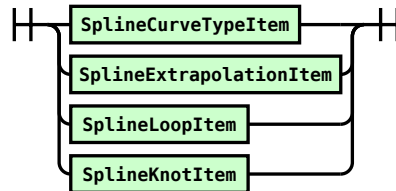
```
SplineTangent <- LeftParen (InlinePadding)* SplineTangentContent (InlinePadding)* RightParen
```



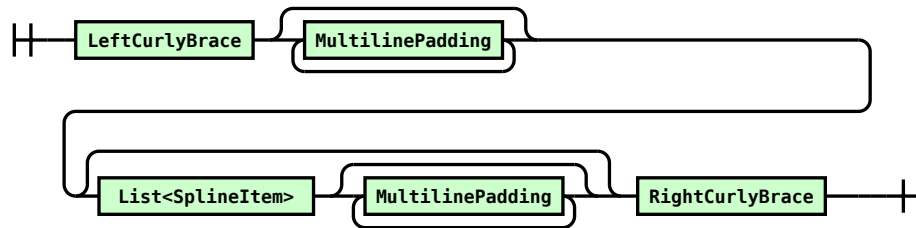
```
SplineTangentContent <- Number (InlinePadding)* Comma (InlinePadding)* Number /
                        Number
```



```
SplineItem <- SplineCurveTypeItem /
              SplineExtrapolationItem /
              SplineLoopItem /
              SplineKnotItem
```



```
SplineMap <- LeftCurlyBrace (MultilinePadding)* ↵
              (List<SplineItem> (MultilinePadding)*)? RightCurlyBrace
```



16.2.14 A General Note on Listops

A listop value for a spec's metadata field contains four separate lists:

- explicit values
- append values
- prepend values
- delete values

If a listop type is not specified when encountering productions containing list op values, those values are implicitly assigned to the explicit list of the list op. If a list op type is not

explicitly specified, it's value is []. Furthermore, unless otherwise specified, values encountered for the same metadata field for the same list op are interpreted as “last one wins”.

Consider the following scenario:

```
#usda 1.0
(  
  foo = [6, 5, 4]  
  append foo = [3, 2, 1]  
  append foo = [1, 2, 3]  
)
```

In this case, the metadata field `foo` will have a listop assigned where each of the list op types have the following values:

- explicit: [6, 5, 4]
- append: [1, 2, 3]
- prepend: []
- delete: []

Empty list ops do not have to be assigned as the empty list is their default value. Except in the case of `attributes`, a value of `None` is equivalent to [].

A list op without an assignment may be treated as a no-op. For example:

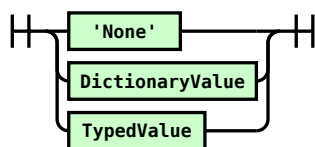
```
#usda 1.0
(  
  append rel myRel  
)
```

Although a legal statement with regard to the grammar, no semantic meaning is assigned to this statement.

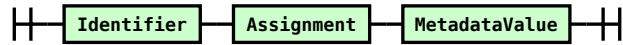
16.2.15 Common Metadata

Metadata can be attached to layers, prims, attributes, and relationships. Although the allowed metadata varies between them, there are some metadata definitions that are shared amongst them. These productions are captured in this section. Additional metadata items that are specific to a particular spec are defined in that specific section.

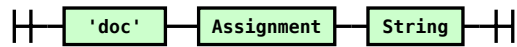
```
MetadataValue <- 'None' /  
                DictionaryValue /  
                TypedValue
```



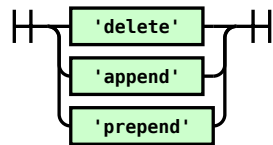
```
KeyValueMetadata <- Identifier Assignment MetadataValue
```



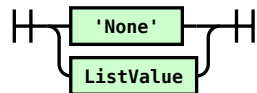
```
DocMetadata <- 'doc' Assignment String
```



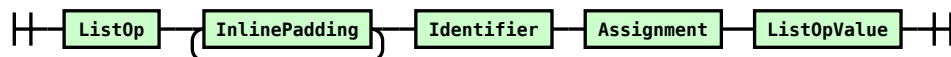
```
ListOp <- 'delete' /
          'append' /
          'prepend'
```



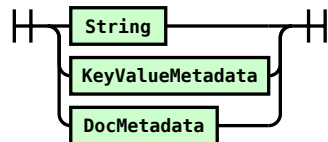
```
ListOpValue <- 'None' /
               ListValue
```



```
ListOpMetadata <- ListOp (InlinePadding)+ Identifier Assignment ListOpValue
```



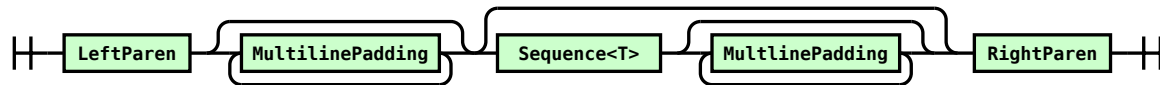
```
SharedMetadata <- String /
                  KeyValueMetadata /
                  DocMetadata
```



Note: In cases where list ops are used in metadata, often there is a choice between a list op and a single item. The use of a single item in place of an explicit list op is always considered to be implicitly declared as an explicit list op.

In addition, since metadata appears in multiple spec declarations, a templated rule is used to make reading those production rules easier:

```
MetadataBlock<T> <- LeftParen (MultilinePadding)* (Sequence<T> (MultilinePadding)*)? RightParen
```



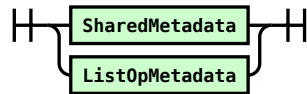
16.2.16 Attribute Specs

Attributes are properties within a prim meant for authoring and retrieving numeric, string, and array valued data, sampled over time.

Note: Although *AttributeType* is represented by an *Identifier* in production rules, an extra layer of validation is done on these types to ensure the *Identifier* reduction results in a valid *type*. The behavior is undefined if the type is unresolvable.

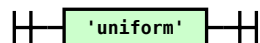
Note: If an *AttributeVariability* is not explicitly specified, the attribute will default to *varying*.

```
AttributeMetadataItem <- SharedMetadata /  
                          ListOpMetadata
```

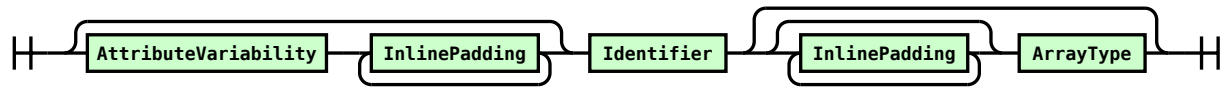


```
AttributeMetadata <- MetadataBlock<AttributeMetadataItem>
```

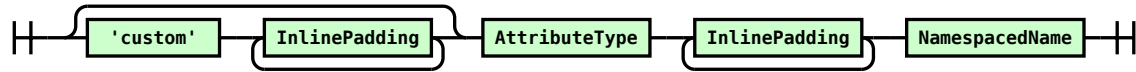
```
AttributeVariability <- 'uniform'
```



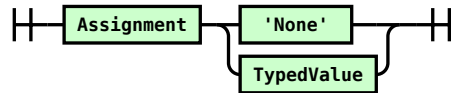
```
AttributeType <- (AttributeVariability (InlinePadding)+)? Identifier ((InlinePadding)* ArrayType)?
```



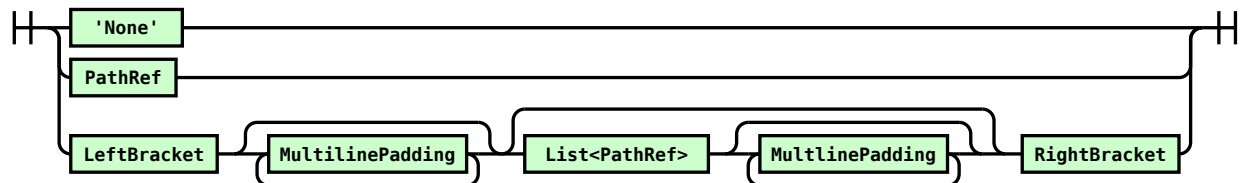
```
AttributeDeclaration <- ('custom' (InlinePadding)+)? AttributeType (InlinePadding)+ NamespacedName
```



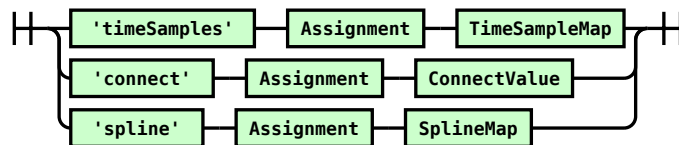
```
AttributeAssignment <- Assignment 'None' /
                        Assignment TypedValue
```



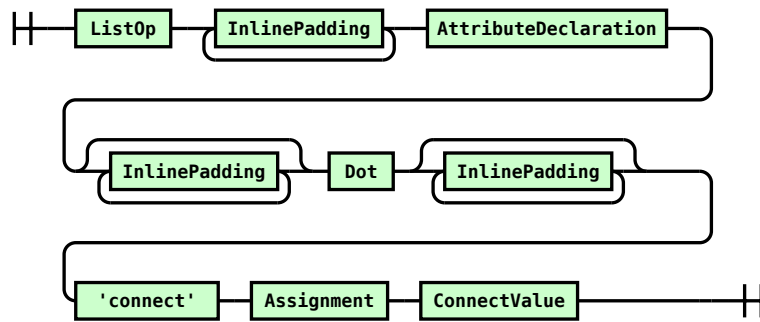
```
ConnectValue <- 'None' /
                PathRef /
                LeftBracket (MultilinePadding)* (List<PathRef> (MultilinePadding)*)? RightBracket
```



```
SpecialAttributeType <- 'timeSamples' Assignment TimeSampleMap /
                        'connect' Assignment ConnectValue /
                        'spline' Assignment SplineMap
```



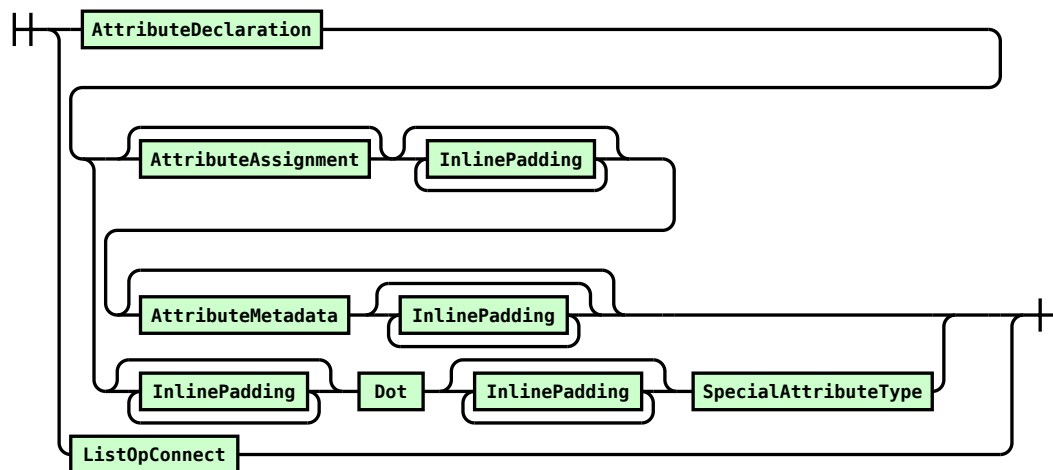
```
ListOpConnect <- ListOp (InlinePadding)+ AttributeDeclaration ↵
                  (InlinePadding)* Dot (InlinePadding)* ↵
                  'connect' Assignment ConnectValue
```



```

AttributeSpec <- AttributeDeclaration <-
  (AttributeAssignment)? (InlinePadding)* <-
  (AttributeMetadata (InlinePadding)*)? /
AttributeDeclaration <-
  (InlinePadding)* Dot (InlinePadding)* SpecialAttributeType /
ListOpConnect

```



Note: Connect attribute list op syntax shares the start of the production (the list op) with other productions at the same level (e.g., list op relationships). In order to keep the grammar readable, these productions are formed as distinct from each other, but implementations will likely want to consume list op tokens as greedily as possible and make a decision about the next lexical tokens that make sense in context to avoid excessive backtracking.

16.2.16.1 Attribute Declarations

The AttributeDeclaration production contains all the information necessary to form the new attribute spec. The items within the declaration are mapped to the following metadata fields on that attribute spec:

- The required metadata field `custom` shall be authored to `true` if the custom keyword was part of the declaration, and authored to `false` otherwise.
- The required metadata field `variability` shall be authored to `uniform` if the keyword is present, and authored to `varying` otherwise.
- The attribute type (it's identifier and array type declaration) is assigned to the `typeName` field.
- The name of the identifier is used to form the path that identifies the new attribute spec.

The path for an attribute is constructed by taking the path identifying the parent spec (i.e. the spec for which this attribute will be one of the `propertyChildren`) and joining it with the parsed namespaced name by means of a `.` character.

For example:

```
def "foo" {
  int bar                                // a new attribute spec with path `/foo.bar`,
                                         // `typeName` = `int`,
                                         // and default variability
  uniform float3d baz:foobar             // a new attribute spec with path `/foo.baz:foobar`,
                                         // `typeName` = `float3d`
                                         // and `variability` = `uniform`
}
```

16.2.16.2 Attribute Values

The value of an attribute assignment that isn't associated with a set of time samples, connections, or splines shall be assigned to the default metadata field. This value is by default unblocked. If an attribute spec is assigned a value of `None`, this indicates that the attribute shall be blocked, thus excluding weaker opinions. The file format is only responsible for setting the metadata field to the value block sentinel in this case; the process of value resolution is responsible for resolving the final value taking the block into consideration. That is:

```
int x = 7      # Value resolves to 7, implicitly blocks any authored weaker opinions and fallback
int x          # Value resolves to authored weaker opinions or fallback if not authored
int x = None   # Skip authored weaker opinions and only resolve to fallback
```

Note that it is not the parser's responsibility to resolve the final value, only to set the metadata fields appropriately such that it can be resolved correctly downstream.

16.2.16.3 TimeSamples

Attribute values that are assigned as part of a time sample map are assigned to the `timeSamples` metadata field. An attribute can have both an assigned value for the default metadata field as well as a value assigned for the `timeSamples` metadata field. Time sample values inside the parsed time sample map are not required to be stored in any explicit order. Individual time samples can have their value blocked if assigned a value of `None`, which has the same semantics as when blocking the default attribute value.

16.2.16.4 Connections

The value of an attribute may also be a set of connections to other prims or properties. These connections are represented by a set of paths, and values parsed from the Connect-Value attribute shall be assigned to the `connectionPaths` metadata field of the attribute spec. Note that this value is a listop, and so can contain any number of paths explicitly set, appended, prepended, or deleted. If no list op was parsed as part of parsing the connect attribute, the list op is assumed to be explicit.

16.2.16.5 Splines

An attribute value may also be associated with a spline. The data for a spline value are built up from each individual spline item contained within the assigned spline map. These can be of the following types:

Curve Type

This string denotes the type of the spline, either `hermite` or `bezier`

Extrapolation

An extrapolation item may either indicate a pre extrapolation or a post extrapolation. The extrapolation type can be one of:

- `none`
- `held`
- `linear`
- `sloped`
- `loop`

If the value is `sloped` there is an additional value that indicates the value of the slope. If the value is `loop`, there is an additional value that indicates whether the loop extrapolation should be `repeat`, `reset`, or `oscillate`.

Loop

A loop item consists of five values mapped to spline data in the following order:

- `prototype start`
- `prototype end`
- `number of pre loop intervals`
- `number of post loop intervals`
- `value offset`

Knots

A spline may have any number of knots specified, where each knot is associated with a time and a value (and an optional pre-value). If two values are specified (separated by an `&`) for the knot value, the first indicates the pre-value and the second the value. A knot may also have a set of parameters that can indicate:

- A knot's pre-tangent

- A knot's post-tangent
- A dictionary of custom data to associate with the knot.

The pre-tangent and post-tangent values indicate either a single value or a set of two values. If single-valued, this is the value of the slope of the tangent. If dual-valued, the values describe the slope of the tangent and the width of the tangent. A post-tangent may also have an associated spline interpolation mode, which can be one of the following:

- none
- held
- linear
- curve

For any of the above values, it is possible from the grammar to see two spline items that describe the same thing. For example:

```
def "foo" {
  float mySpline.spline = {
    bezier,
    hermite
  }
}
```

In the above example, a `curve` type item is parsed twice for the same spline, with the first item indicating the spline is `bezier` and the second item indicating the spline is `hermite`. In all cases where multiple values for the same item type are parsed, the last one the parser encounters is the one for which the values of the spline are stored. In the case of knots, a spline may have multiple knots, but within the same knot duplicate information may be specified. In these cases, the last value wins. Consider the following:

```
def "foo" {
  float mySpline.spline = {
    6: 6532.62342; post curve (3.6, -7.8); post none,
  }
}
```

In the above example, the knot at time 6 has two items defining its post-tangent value. In this case, the knot will be assigned a post-tangent value of `none` with no associated slope.

16.2.16.6 Attribute Types and Value Validation

Attribute types are represented as identifiers (with an optional `[]` indicating whether it's an array or not).

The parser must ensure that if a type is resolvable (i.e., it is a known type specified in [Foundational Data Types](#)), that the value specified for the attribute is consistent with its declared type.

If the inferred value parsed is not consistent with the declared type, an attempt may be made to coerce the value to be consistent with the declared type. Values that are inconsis-

tent with the declared type, and that cannot be coerced to the type, must result in an error. If a type is unresolvable, the behavior for value validation is undefined.

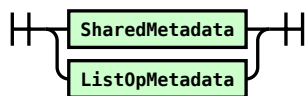
A few examples:

```
def "foo" {
  int bar = 3 // ok - value consistent with the declared type
  int[] baz = 3 // error - scalar value for attribute declared
                // as vectorized type
  float foobar = 4 // ok - parsed value can be coerced to be consistent
                   // with declared type (e.g., 4.0)
  point3d[] points = [(-120.5, 181.73, 112.3)] // ok - value consistent with the underlying
                                                // type `double3` of the semantic alias `point3d`
  double4 zero = (0.0, 0.0, 0.0) // error - ill-dimensioned tuple value
  matrix4d transform = [
    (0.9, 0, 0, 0),
    (0, 0.9, 0, 0),
    (0, 0, 0.9, 0),
    (0, 0, 0, 1)
  ] // error - value is array-valued rather than tuple-valued
  myType foobaz = "some value" // undefined - myType unresolvable
}
```

16.2.16.7 Relationship Specs

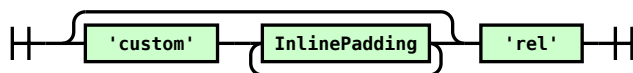
A *relationship* is a property that creates a dependency between scene graph objects by allowing a prim to target other prims, attributes, or relationships.

```
RelationshipMetadataItem <- SharedMetadata /
                          ListOpMetadata
```

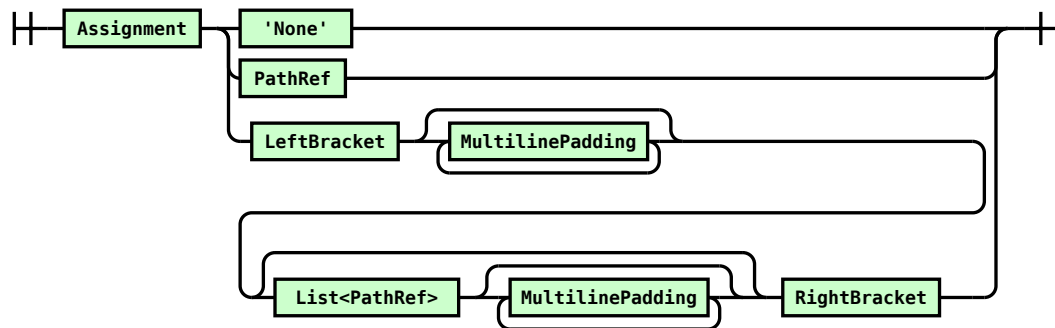


```
RelationshipMetadata <- MetadataBlock<RelationshipMetadataItem>
```

```
RelationshipType <- 'rel' /
                  'custom' (InlinePadding)+ 'rel'
```

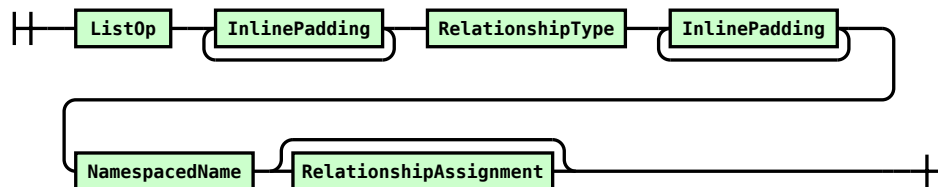


```
RelationshipAssignment <- Assignment 'None' /
                      Assignment PathRef /
                      Assignment LeftBracket (MultilinePadding)* ↵
                      (List<PathRef> (MultilinePadding)*)? RightBracket
```



```

ListOpRelationship <- ListOp (InlinePadding)+ RelationshipType (InlinePadding)+ ↵
                        NamespacedName (RelationshipAssignment)?
  
```



```

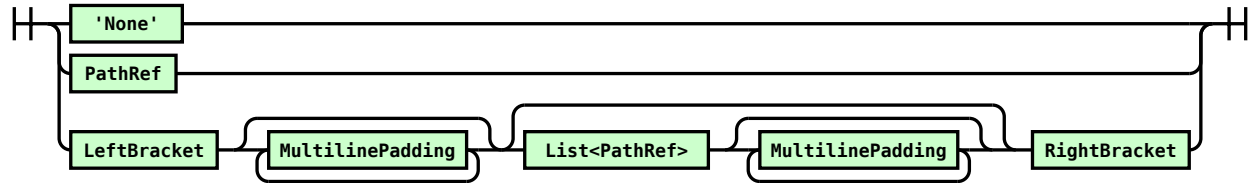
RelationshipSpec <- RelationshipType (InlinePadding)+ NamespacedName ↵
                    (RelationshipAssignment)? (InlinePadding)* (RelationshipMetadata)? /
RelationshipType (InlinePadding)+ NamespacedName ↵
(InlinePadding)? LeftBracket ↵
    (InlinePadding)* PathRef (InlinePadding)* ↵
RightBracket /
RelationshipType (InlinePadding)+ NamespacedName ↵
(InlinePadding)* Dot (InlinePadding)* TimeSampleOrDefault /
ListOpRelationship
  
```



```

InheritsOrSpecializesList <- 'None' /
    PathRef /
    LeftBracket (MultilinePadding)* (List<PathRef> (MultilinePadding)*)?
    RightBracket

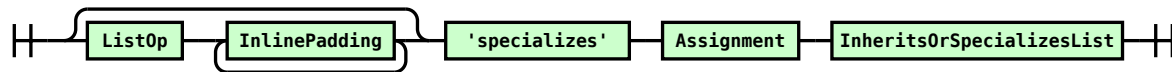
```



```

SpecializesMetadata <- (ListOp (InlinePadding)+)? 'specializes' Assignment InheritsOrSpecializesList

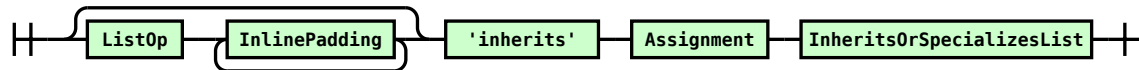
```



```

InheritsMetadata <- (ListOp (InlinePadding)+)? 'inherits' Assignment InheritsOrSpecializesList

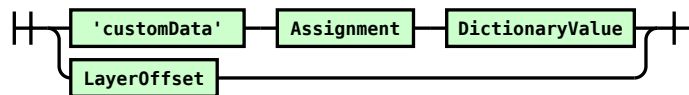
```



```

ReferenceParameter <- 'customData' Assignment DictionaryValue /
    LayerOffset

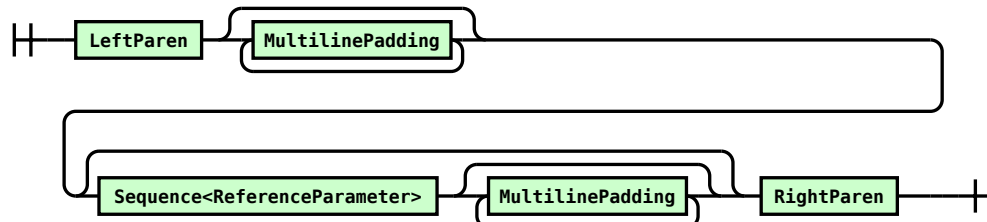
```



```

ReferenceParameters <- LeftParen (MultilinePadding)* ↵
    (Sequence<ReferenceParameter> (MultilinePadding)*)? RightParen

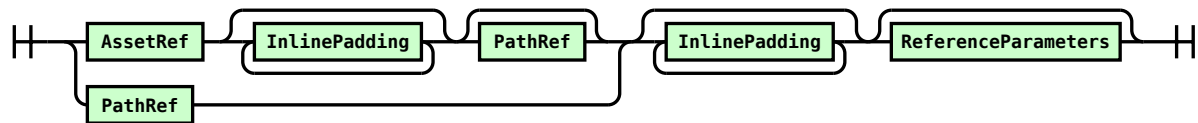
```



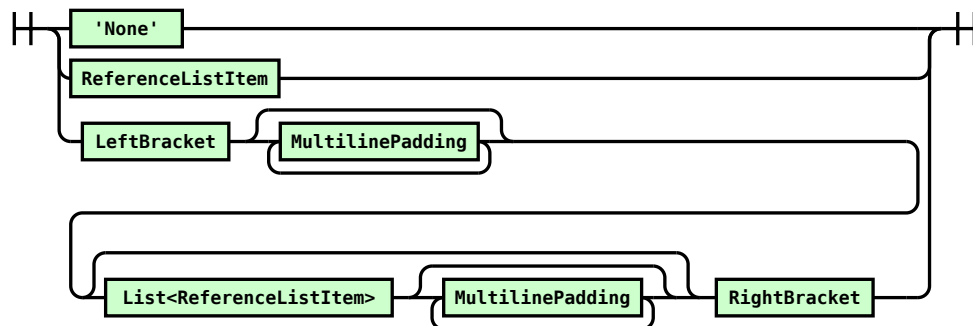
```

ReferenceListItem <- AssetRef (InlinePadding)* (PathRef)? (InlinePadding)* (ReferenceParameters)? /
    PathRef (InlinePadding)* (ReferenceParameters)?

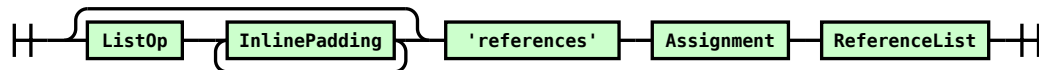
```



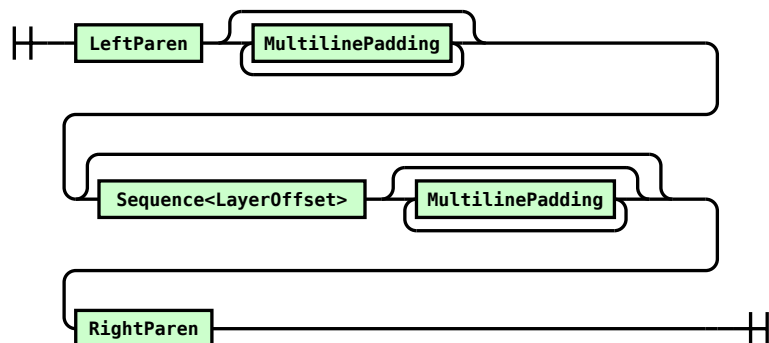
```
ReferenceList <- 'None' /
  ReferenceListItem /
  LeftBracket (MultilinePadding)* ↵
  (List<ReferenceListItem> (MultilinePadding)*)? RightBracket
```



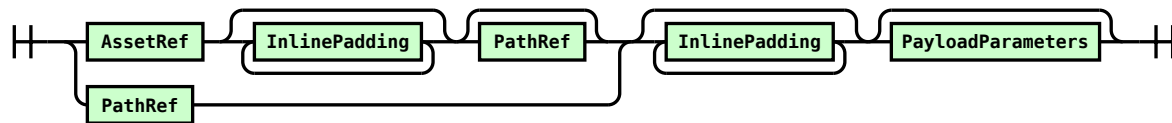
```
ReferencesMetadata <- (ListOp (InlinePadding)+)? 'references' Assignment ReferenceList
```



```
PayloadParameters <- LeftParen (MultilinePadding)* ↵
  (Sequence<LayerOffset> (MultilinePadding)*)? ↵
  RightParen
```

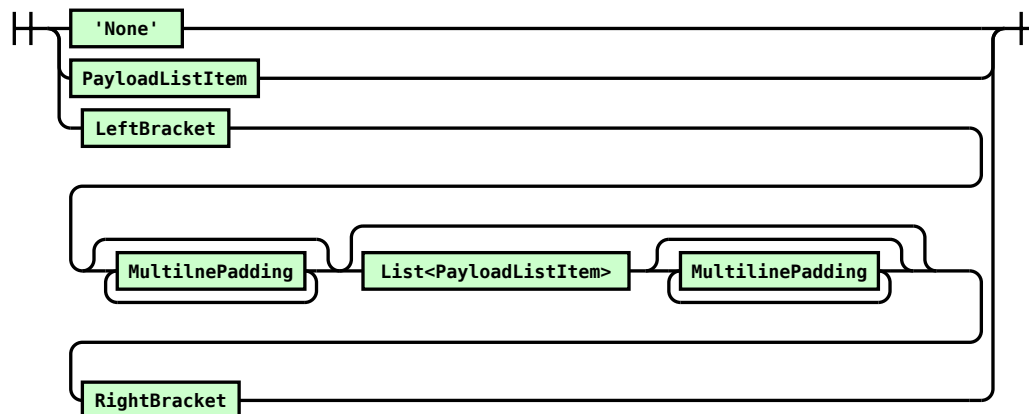


```
PayloadListItem <- AssetRef (InlinePadding)* (PathRef)? (InlinePadding)* (PayloadParameters)? /
  PathRef (InlinePadding)* (PayloadParameters)?
```



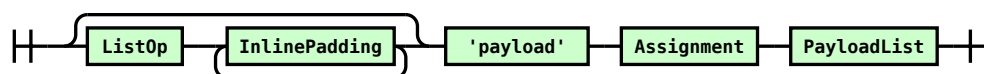
```

PayloadList <- 'None' /
               PayloadListItem /
               LeftBracket <-
               (MultilinePadding)* (List<PayloadListItem> (MultilinePadding)*)? <-
               RightBracket
  
```



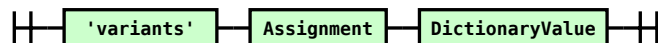
```

PayloadMetadata <- (ListOp (InlinePadding)+)? 'payload' Assignment PayloadList
  
```



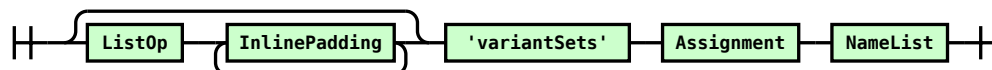
```

VariantsMetadata <- 'variants' Assignment DictionaryValue
  
```



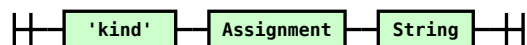
```

VariantSetsMetadata <- (ListOp (InlinePadding)+)? 'variantSets' Assignment NameList
  
```



```

KindMetadata <- 'kind' Assignment String
  
```



```

PrimMetadataItem <- SharedMetadata /
    ListOpMetadata /
    KindMetadata /
    PayloadMetadata /
    InheritsMetadata /
    SpecializesMetadata /
    ReferencesMetadata /
    VariantsMetadata /
    VariantSetsMetadata

```

```

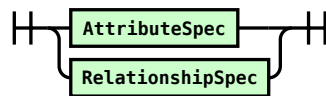
PrimMetadata <- MetadataBlock<PrimMetadataItem>

```

```

PropertySpec <- AttributeSpec / RelationshipSpec

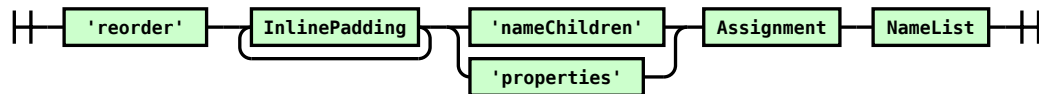
```



```

ChildOrPropertyOrderStatement <- 'reorder' (InlinePadding)+ 'nameChildren' Assignment NameList /
    'reorder' (InlinePadding)+ 'properties' Assignment NameList

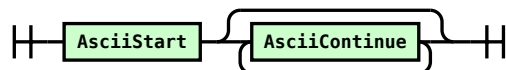
```



```

PrimTypeName <- AsciiStart (AsciiContinue)*

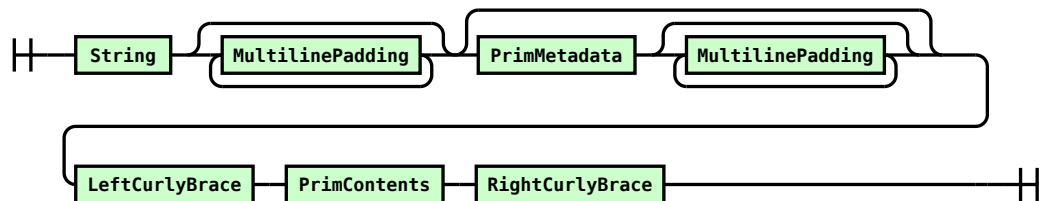
```



```

VariantStatement <- String (MultilinePadding)* ((PrimMetadata) (MultilinePadding)*)? ↵
    LeftCurlyBrace PrimContents RightCurlyBrace

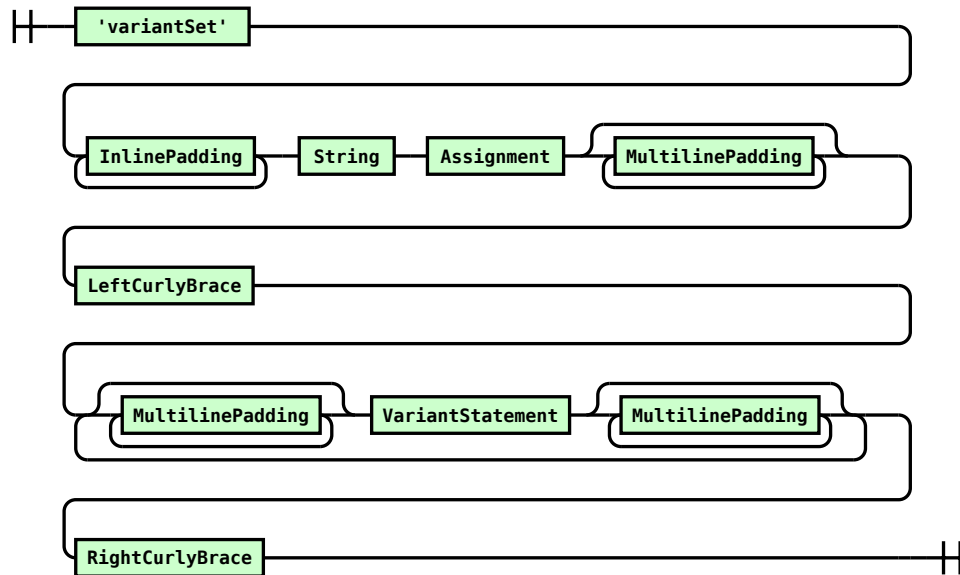
```



```

VariantSetStatement <- 'variantSet' ↵
    (InlinePadding)+ String Assignment (MultilinePadding)* ↵
    LeftCurlyBrace ↵
    ((MultilinePadding)* VariantStatement (MultilinePadding)*)+ ↵
    RightCurlyBrace

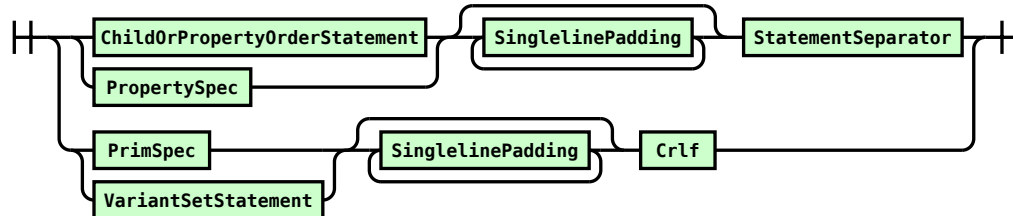
```



```

PrimItem <- ChildOrPropertyOrderStatement (SinglelinePadding)* StatementSeparator /
  PropertySpec (SinglelinePadding)* StatementSeparator /
  PrimSpec (SinglelinePadding)* Crlf /
  VariantSetStatement (SinglelinePadding)* Crlf

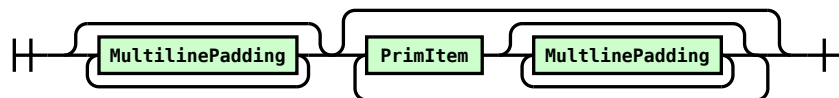
```



```

PrimContents <- (MultilinePadding)* (PrimItem (MultilinePadding)*)*

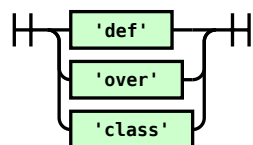
```



```

PrimSpecifier <- 'def' /
  'over' /
  'class'

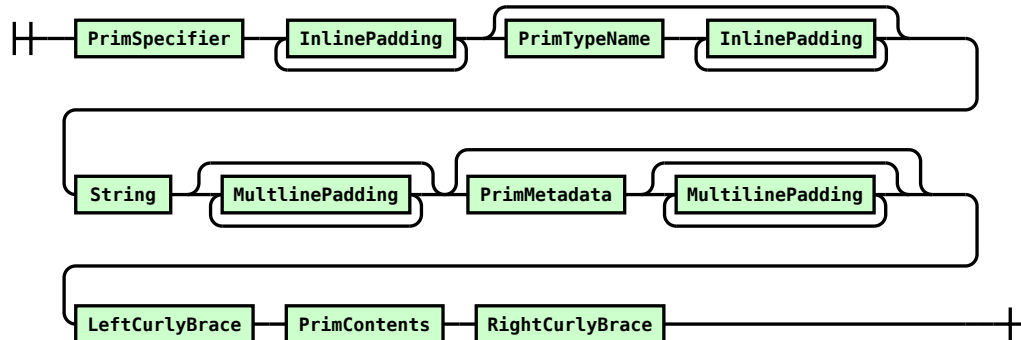
```




```

PrimSpec <- PrimSpecifier (InlinePadding)+ (PrimTypeName (InlinePadding)+)? ↵
          String (MultilinePadding)* (PrimMetadata (MultilinePadding)+)? ↵
          LeftCurlyBrace PrimContents RightCurlyBrace

```



16.2.17.1 Prim Declaration Components

The value parsed for the prim specifier (e.g. `def`, `over`, or `class`) shall be placed in the specifier metadata field of the document data model.

The value parsed for the prim type name shall be placed in the `typeName` metadata field of the document data model. If no type name is present, this value shall not be assigned.

The value parsed for the prim identifier (i.e., the `String` production denoting the name of the prim spec) shall be appended to the path of the prim spec's parent to form the uniquely addressable path for the prim spec.

For example, parsing the definition for the following prim specs:

```
#usda 1.0
```

```

def Foo "bar" {
  over "Baz" {
  }
}

```

shall result in creation of a new prim spec with uniquely addressable path `/bar` with the following metadata fields assigned:

- specifier: `def`
- typeName: `Foo`

and a new prim spec with uniquely addressable path `/bar/Baz` with the following metadata fields assigned:

- specifier: `over`

16.2.17.2 Prim Spec Children

A prim spec can contain three different kinds of children:

- Prim specs
- Property specs
- VariantSet specs

Each of these categories of items are assigned to different metadata fields of the prim spec in the data model:

- `primChildren` (the list of names of the prim specs contained within this prim spec)
- `propertyChildren` (the list of names of the property specs contained within this prim spec)
- `variantSetChildren` (the list of names of the variantSet specs contained within this prim spec)

Note that in each case, the names assigned to the list are the simple names of the spec rather than the full path of the spec. For examples, consider the following case:

#usda 1.0

```
def "foo" {
  int bar = 4
  over "baz" {
  }

  variantSet "variant" = {
    "x" {
      def "variantFoo" {
      }
    }
  }
}
```

This would result in the following set of named specs with metadata fields (only relevant fields shown):

- `/foo` (prim spec)
 - `primChildren`: [baz]
 - `propertyChildren`: [bar]
 - `variantSetChildren`: [variant]
- `/foo.bar` (attribute spec)
- `/foo/baz` (prim spec)
- `/foo{variant}` (variantSet spec)

16.2.17.2.1 specifier and typeName for Variants

Variant specs have the same set of allowed and required metadata fields as prim specs. The document model states that `typeName` may be authored and `specifier` is required to be authored for variants. The usda format does not currently provide for the storage of either field and should always report `typeName` as unauthored and `specifier` as being authored with value over until a future revision can specify explicit storage.

16.2.17.3 Child Ordering

Prim specs have a notion of ordering of their prim spec children and property spec children for traversal operations. If no explicit ordering is provided, traversal occurs via the `primChildren` and `propertyChildren` metadata fields. If ordering is explicitly specified (via `reorder nameChildren` for prim specs and `reorder propertyChildren` for property specs), the list of names specified are added to the `primOrder` and `propertyOrder` metadata fields in the data model of the prim spec being parsed. These lists may be sparse (i.e., contain only a few names as opposed to all children) and if encountered multiple times when parsing a prim spec will overwrite the previous value.

Consider the following example:

```
#usda 1.0
```

```
def "foo" {  
  int bar = 4  
  over "baz" {  
  }  
  def "foobar" {  
  }  
  string foobaz = "value"  
  float barbaz = 3.7  
  
  reorder nameChildren = ["foobar"]  
  reorder propertyChildren = ["barbaz"]  
}
```

This would result in the following children relevant metadata fields for the spec `/foo`:

- `primChildren`: [baz, foobar]
- `propertyChildren`: [bar, foobaz, barbaz]
- `primOrder`: [foobar]
- `propertyOrder`: [barbaz]

implying that when traversing the prim children of `/foo`, `/foo/foobar` should be traversed before `/foo/baz` and that when traversing the property children of `/foo`, `/foo.barbaz` should be traversed before `/foo.bar` and `/foo.foobaz`.

16.2.17.4 Prim Metadata

Prim metadata comprises the largest group of metadata of all the specs. The majority of these metadata productions are aimed at providing opinions for the composition system. Each of these items is mapped to metadata fields as follows:

- Payload Metadata - values are assigned to the payload metadata field
- References Metadata - values are assigned to the references metadata field
- Inherits Metadata - values are assigned to the `inheritPaths` metadata field
- Specializes Metadata - values are assigned to the specializes metadata field

- VariantSets Metadata - values are assigned to the variantSetNames metadata field
- Variants Metadata - values are assigned to the variantSelection metadata field
- Kind Metadata - value is assigned to the kind metadata field

Several well-known metadata fields for prim specs (e.g., active, instanceable, displayName, etc.) do not have concretely specified parser production rules and instead are set via the generic key / value data of KeyValueMetadata.

Note that all metadata fields can be seen more than once - in these cases the last value is that which is ultimately assigned to the prim spec metadata field. Also note that seeing a particular metadata production in listop form and non-listop form is different from seeing the metadata production in e.g., the same listop form or non-listop form multiple times. For example:

```
def "foo" (
  payload = </bar>
  prepend payload = </baz>
){
}

def "foobar" (
  payload = </bar>
  prepend payload = </baz>
  prepend payload = </foo>
){
}
```

Here, the prim spec /foo has two distinct items of payload metadata, an explicit listop and a prepend listop. /foobar on the other hand has a duplicate metadata item (i.e., two prepend listops).

In this case, the list for the prepend list op stored on the payload metadata field for /foobar should be [/foo].

16.2.17.5 Reference vs Payload Parameters

Note that the production rules for reference and payload parameters both contain information about the layer offset (e.g., offset and scale). The final layer offset is built up by parsing each item and replacing the value for either offset or scale in the final structure instance. However, reference parameters can have an additional value mixed in - custom-Data. Implementations shall parse this value out, but there is no element of the structure capturing a reference nor a specific metadata field in the prim spec that captures this data and hence the value can be discarded.

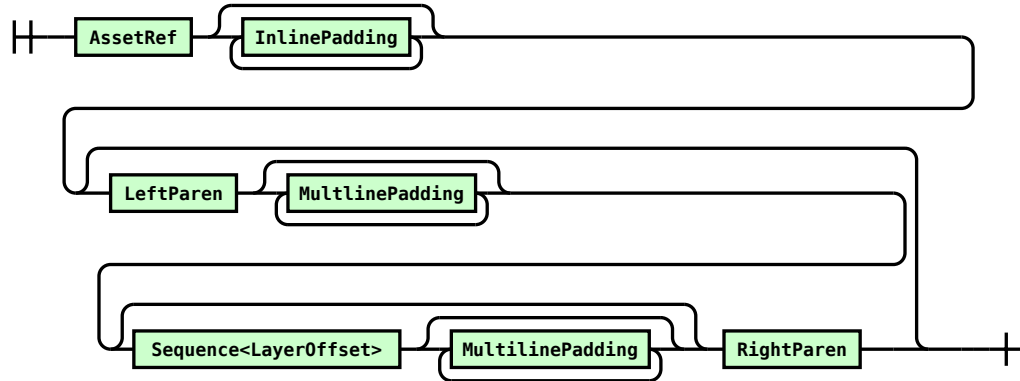
16.2.18 Layer

The *layer* serves as the representation of the document data model and can be persisted in different formats. This part of the specification describes a text scene description format with the suffix **.usda**.

```

SublayerItem <- AssetRef (InlinePadding)* ↵
                (LeftParen (MultilinePadding)* ↵
                (Sequence<LayerOffset> (MultilinePadding)*)?
                RightParen)?

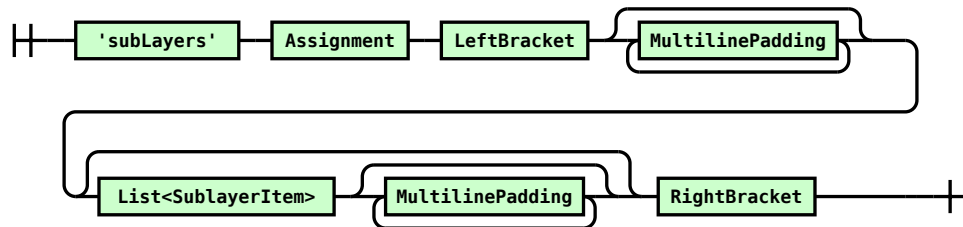
```



```

SublayerMetadata <- 'subLayers' Assignment LeftBracket (MultilinePadding)* ↵
                    (List<SublayerItem> (MultilinePadding)*)? RightBracket

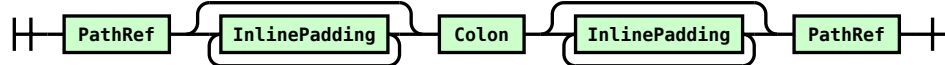
```



```

RelocatesItem <- PathRef (InlinePadding)* Colon (InlinePadding)* PathRef

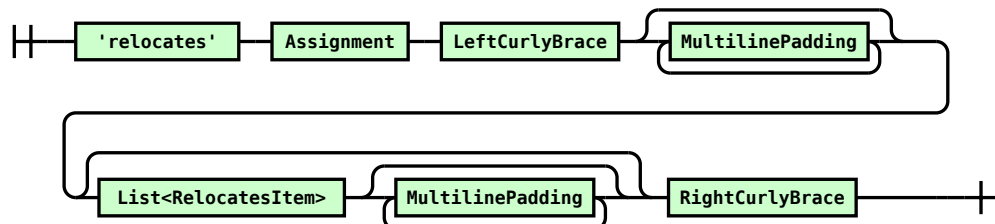
```



```

RelocatesMetadata <- 'relocates' Assignment LeftCurlyBrace (MultilinePadding)* ↵
                    (List<RelocatesItem> (MultilinePadding)*)? RightCurlyBrace

```



```

LayerMetadataItem <- SharedMetadata /
                      ListOpMetadata /
                      SublayerMetadata /
                      RelocatesMetadata

```

```

LayerMetadata <- MetadataBlock<LayerMetadataItem>

```

```

LayerHeader <- Pound 'usda' (InlinePadding)* (Digit)+ Dot (Digit)+ (Dot (Digit))+?

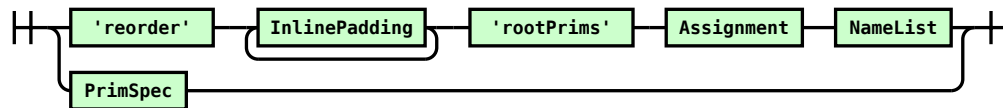
```



```

LayerItem <- 'reorder' (InlinePadding)+ 'rootPrims' Assignment NameList / PrimSpec

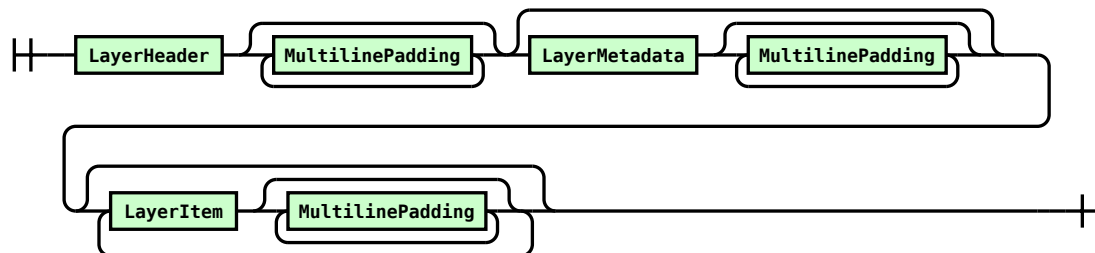
```



```

LayerSpec <- LayerHeader (MultilinePadding)* (LayerMetadata (MultilinePadding)*)? ↵
              (LayerItem (MultilinePadding))*

```



Note: This section describes the 1.0 version of the USDA document model. Any files based on this specification shall have a layer header of the form #usda 1.0

16.2.18.1 Layer Header

The Digit elements of the LayerHeader production correspond, in order to:

- The MAJOR version of the format
- The MINOR version of the format
- The (optional) PATCH version of the format

There is no data model mapping for the layer header. Validation is done within the parser itself and can be discarded.

16.2.18.2 Layer Content

Each spec contained within a layer is uniquely addressable via its path. If at any point a spec definition is encountered with a path that has been previously encountered, an error must be emitted and all processing shall cease. For example:

```
#usda 1.0

def "foo" {
}

def "bar" {
}

over "foo" {
}
```

In the above case, the spec at path \foo is encountered a second time (with a different specifier). As such, the layer is invalid and no further processing takes place.

16.2.18.3 Sublayer Metadata

Sublayer metadata maps to the following document data model metadata fields:

- subLayers (the asset refs)
- subLayerOffsets (the layer offsets)

These fields must be of the same length. If no layer offset is specified for a sublayer item (asset ref), it's corresponding entry in the subLayerOffsets list must receive a default instance of the subLayerOffsets structure.

Although it is syntactically allowed to specify e.g. multiple scale fields for a layer offset associated with an asset ref, the last value specified shall be the value used in the subLayerOffsets structure. For example, consider the following content:

```
subLayers = [
  @myAsset@ (
    scale = 2.3
    offset = 3.0
    scale = 4.1
  )
]
```

Scale is specified twice in this definition, but the final subLayerOffsets structure that should be written must have an offset of 3.0 and scale of 4.1.

16.2.18.4 Root Prim Reordering

A layer item can be either a prim spec or a reorder rootPrims statement. A reorder rootPrims statement maps to the primOrder metadata field. Similarly to subLayerOffsets, syntactically it is possible to have multiple reorder rootPrims statements, each of which could

have the full list of root prims in the layer or some sparse subset. These lists shall not be combined - the last value specified shall be the value used. For example:

```
reorder rootPrims = ["baz"]
```

```
def "foo" {  
}
```

```
def "bar" {  
}
```

```
def "baz" {  
}
```

```
reorder rootPrims = ["bar", "baz"]
```

shall result in the `primOrder` field containing `bar` and `baz` in that order.

16.2.18.5 Layer Relocates Metadata

A prim path can be relocated (i.e. renamed or reparented) in the scope of the local layer namespace. The source of the composition arc remains the same (that is, the prim spec to which the source path refers is still used as normal in composition), but may be referred to in the local layer namespace by a new name or with a new parent. Each relocate item is defined by a source prim path and target prim path, and a list of these are stored in the `layerRelocates` field of the layer spec. For example, assume there is a prim `baz` in a layer:

```
def "foo" {  
  def "bar" {  
    def "baz" {  
    }  
  }  
}
```

and that the layer is referenced into another layer:

```
def "layerdefaultprim" (  
  references = @barlayer.usda@</foo>  
)  
{  
}
```

A relocates mapping can be provided such that in the second layer, we can rename `baz` to `barbaz` and reparent it to `layerdefaultprim` then create an over for that prim:

```
#usda 1.0
```

```
(  
  relocates = {
```



```

        </layerdefaultprim/bar/baz> : </layerdefaultprim/barbaz>
    }
)

def "layerdefaultprim" (
    references = @barlayer.usda@</foo>
)
{
    over "barbaz" {
        int bazattribute = 5
    }
}

```

Note that the parser must not attempt to validate whether a pair of relocate paths are valid or not as it does not have all the information to do so - this must be done as part of composition.

16.2.19 Compatibility with Legacy Content

The grammar defined above represents the normative specification for the USDA grammar. However, as the OpenUSD ecosystem has evolved over the years, legacy content predating this specification exists. To provide maximum compatibility with this content, implementations have the option of supporting additional production rules for the USDA grammar. The semantics of these rules are left unspecified; the additional production rules represent only a means to successfully parse legacy content for compatibility. Many of the production rules described in this section are additional rules while some rules are modified from the normative rules.

```

LegacyKeyword <- 'add' /
                  'config' /
                  'displayUnit' /
                  'permission' /
                  'prefixSubstitutions' /
                  'reorder' /
                  'suffixSubstitutions' /
                  'symmetryArguments' /
                  'symmetryFunction'

Keyword <- 'append' /
           'class' /
           'connect' /
           'custom' /
           'customData' /
           'default' /
           'def' /
           'delete' /
           'dictionary' /
           'doc' /

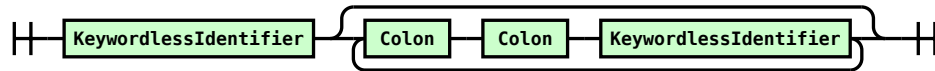
```

```

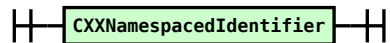
'inherits' /
'kind' /
'nameChildren' /
'None' /
'offset' /
'over' /
'payload' /
'prepend' /
'properties' /
'references' /
'relocates' /
'rel' /
'reorder' /
'rootPrims' /
'scale' /
'subLayers' /
'specializes' /
'timeSamples' /
'uniform' /
'variantSet' /
'variantSets' /
'variants' /
'varying' /
LegacyKeyword

```

```
CXXNamespacedIdentifier <- KeywordlessIdentifier (Colon Colon KeywordlessIdentifier)*
```



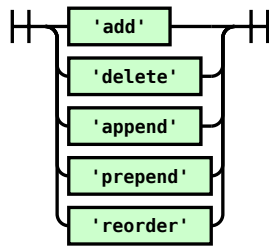
```
Identifier <- CXXNamespacedIdentifier
```



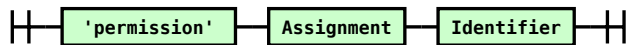
```

ListOp <- 'add' /
         'delete' /
         'append' /
         'prepend' /
         'reorder'

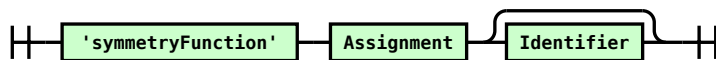
```



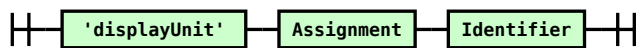
```
PermissionMetadata <- 'permission' Assignment Identifier
```



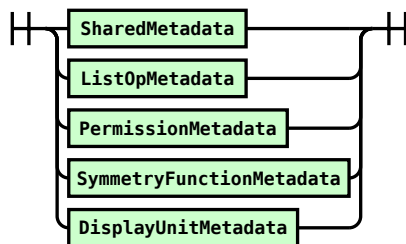
```
SymmetryFunctionMetadata <- 'symmetryFunction' Assignment (Identifier)?
```



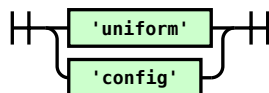
```
DisplayUnitMetadata <- 'displayUnit' Assignment Identifier
```



```
AttributeMetadataItem <- SharedMetadata /
                          ListOpMetadata /
                          PermissionMetadata /
                          SymmetryFunctionMetadata /
                          DisplayUnitMetadata
```



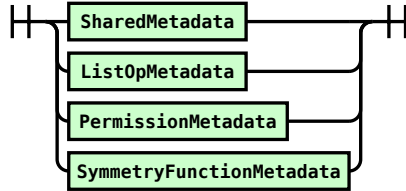
```
AttributeVariability <- 'uniform' / 'config'
```



```

RelationshipMetadataItem <- SharedMetadata /
                             ListOpMetadata /
                             PermissionMetadata /
                             SymmetryFunctionMetadata

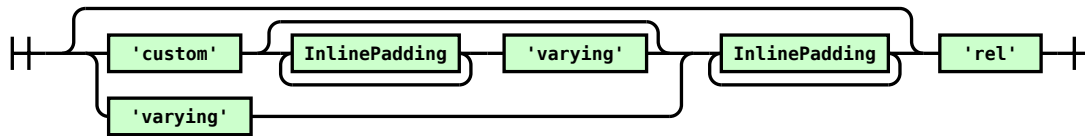
```



```

RelationshipType <- 'rel' /
                    'custom' (InlinePadding)+ 'rel' /
                    'custom' (InlinePadding)+ 'varying' (InlinePadding)+ 'rel' /
                    'varying' (InlinePadding)+ 'rel'

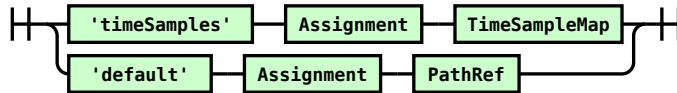
```



```

TimeSampleOrElseDefault <- 'timeSamples' Assignment TimeSampleMap /
                           'default' Assignment PathRef

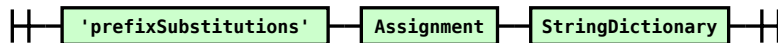
```



```

PrefixSubstitutionsMetadata <- 'prefixSubstitutions' Assignment StringDictionary

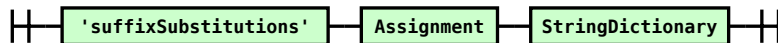
```



```

SuffixSubstitutionsMetadata <- 'suffixSubstitutions' Assignment StringDictionary

```



```

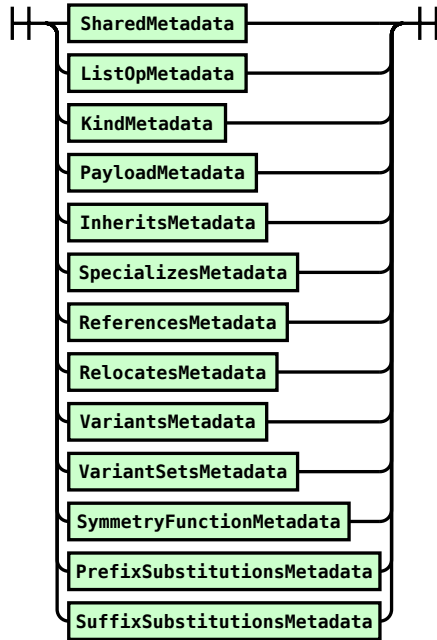
PrimMetadataItem <- SharedMetadata /
                    ListOpMetadata /
                    KindMetadata /
                    PayloadMetadata /
                    InheritsMetadata /

```

```

SpecializesMetadata /
ReferencesMetadata /
RelocatesMetadata /
VariantsMetadata /
VariantSetsMetadata /
SymmetryFunctionMetadata /
PrefixSubstitutionsMetadata /
SuffixSubstitutionsMetadata

```



Note: An `AttributeVariability` of config shall be interpreted as `SdfVariability.Uniform`.

16.2.20 Suggested Simplifications To the Grammar

- Making tuples and arrays homogeneous both improves the performance of the grammar and should make the grammar simpler to express
- Make statement separators consistent (always new lines or always new lines or ;)
- Deprecate C++ style comments in favor of Python style comments
- Deprecate `CXXNamespacedIdentifiers` for typenames
- Minimize metadata statements that require context to parse (i.e. `customData` is just a dictionary, `kind` is just a token, substitutions are just dictionaries)

16.3 Binary

USD uses the Crate format to encode its data in a binary format. They are typically represented with the `.usdc` or `.usd` extension.

16.3.1 Order of Reads

The Crate format is designed for minimal parsing on file load. To achieve this, the general structure of the file is set up in the preamble section below that must be read first. Value types are parsed on demand from there on out.

16.3.2 Endianness

Crate numbers are stored with the least significant bit first to allow for faster loading on Little Endian systems

16.3.3 Alignment

It is strongly recommended that uncompressed arrays be aligned to 8-byte boundaries, as this allows for more efficient reader implementations.

16.3.4 Floating Point

Floating point data in Crate documents use the IEEE-754 specification <https://standards.ieee.org/ieee/754/6210/>

16.3.5 Index

Many fields reference into another section to get their value. USD uses unsigned 32-bit integers as an index unless otherwise specified.

If a section below uses the title case Index or Indices , it implicitly refers to this form of Index.

16.3.6 Check sizes and file offsets

Most binary sections as described below will specify a size prior to reading the value, or an offset within the file to read.

It is recommended that implementations verify that the combination of size and offset does not result in a file read beyond the end of the file.

16.3.7 Compression

16.3.7.1 LZ4 Compression

Various sections and data blocks use LZ4 compression. LZ4 compression is described by the LZ4 library at <https://lz4.org/> and is not expanded here, unless specifics must be provided.

LZ4 compressed data buffers will store the number of chunks as their first element, as an unsigned 64-bit integer.

If the number of chunks is 0, then the buffer is decompressed as a whole.

If the number of chunks is specified as higher than zero, the first byte of each chunk is an unsigned 64-bit integer representing the size of the chunk in bytes. This is followed by the actual chunk data itself.

Description	Size	Value
Number of Chunks	8 bytes	Unsigned 64 bit integer
Chunk Size . Only exists if number of chunks > 0	8 bytes	Unsigned 64 bit integer
Chunk Data	Defined by Chunk Size if number of chunks is more than 0, or the rest of the buffer if there are 0 chunks.	

16.3.7.2 Compressed Integer Arrays

Compressed Integer Arrays are used commonly throughout the Crate format. They can represent either 32-bit or 64-bit integers as required by their point of use.

They encode integers into a compressed array that is further compressed using LZ4.

The integer compression algorithm encodes the array as a variable width series of integers that represent a delta from the previous value.

The array is prefixed by a series of 2 bit codepoints that represent the width of the specific entry in the list.

16.3.7.2.1 Conversion to differences

An array of integers is represented as their difference from the preceding value.

Taking this example array

```
[123, 124, 125, 100125, 100125, 100126, 100126]
```

It is converted to

```
[123, 1, 1, 100000, 0, 1, 0]
```

16.3.7.2.2 Encoding Values

The array of differences is described with a single starting integer, and a series of 2 bit code points that describe the width of each encoded integer in the array. Following these code points, are the values associated with them.

The first integer is the most common value in the array. For the example array, this is 1. This value is stored as a signed version of the integer type in use for the array.

Each value in the array of differences is then converted to a code point and a value relative to the previous value. The value should use the smallest width integer type that can represent the value.

If no previous value exists, it starts at 0.

The values are encoded according to this table

Codepoint	Meaning	Value Encoding
00	Use the Common Value	Discard the value
01	Quarter width signed integer.	Int8 for 32-bit values Int16 for 64-bit values
10	Half width signed integer	Int16 for 32-bit values Int32 for 64-bit values
11	Full width signed integer	Signed representation of full width value

The layout of the resulting array would be as follows

Description	Size	Value
Common Value	Signed Integer of same size	The most common value in the array of differences
Codepoints	2 bits per final value	As described in table above
Padding	Rounds out the codepoints to a full byte boundary	Zeroed bits
Encoded Values		The list of encoded values as described in the table above

For the given sample, this would be the result

```
input = [123, 124, 125, 100125, 100125, 100126, 10026]
```

```
output = [int32(1) 01 00 00 11 01 00 01 XX int8(123) int32(100000) int8(0) int8(0)]
```

16.3.7.2.3 Compression of the Buffer

The start of a compressed integer array is an unsigned 64-bit integer representing the compressed size of the data. This compressed size defines the number of bytes to read immediately afterward that contains the compressed data.

Following this is the compressed data of this size that must be decompressed using the LZ4 algorithm.

Description	Size	Value
Compressed Size	8 bytes	Unsigned 64 bit integer
Data	Defined by Compressed Size	

16.3.8 Preamble

The Crate format has a preamble that must be read prior to parsing the rest of the file.

The preamble contains information about the file version, and information on how to parse the prim hierarchy.

16.3.8.1 Header

Every Crate file starts with a format and version identifier.

Crate files must start with the PXR-USDC format identifier, followed by the version identifiers.

Description	Size	Value
File type identifier	8 bytes	String of value: PXR-USDC Hex of value: 5058522D55534443
Major Version	1 byte	Unsigned 8 bit integer
Minor Version	1 byte	Unsigned 8 bit integer
Patch Version	1 byte	Unsigned 8 bit integer
Unused	5 bytes	

The unused bytes are recommended to be zero value bytes, but are not required to be so currently. They may be used in future versions of the specification for specific semantic meanings.

16.3.8.2 Crate Versions

At the time of publishing of this specification, there are several known versions of the Crate document format.

Any implementer of this specification must support the latest crate version in this specification for reading.

It is recommended, but not required, to use the lowest Crate version possible when writing a Crate document. This allows maximum compatibility with runtimes that may not have updated to support newer format features.

The minimum crate version on write must be 0.8.0, and the maximum must be 0.12.0. Readers do not need to support versions of Crate prior to the minimum version, and this specification does not include specification for those prior versions.

The document will make note of features below that require crate versions newer than this minimum version.

This table includes information about the changes in each Crate version, including those prior to the minimum version.

Version	Description
0.12.0	Added support for Splines
0.11.0	Added support for Relocates in Layer Metadata
0.10.0	Added support for Path Expression value types

Version	Description
0.9.0	Added support for TimeCode and TimeCode Array value types
0.8.0	Added support for PayloadListOp values and Payload values with layer offsets
0.7.0	Array sizes are written as 64 bit integers
0.6.0	Supports compressed (scalar) floating point arrays that are either all ints or can be represented efficiently with a lookup table
0.5.0	Compressed (u)int and (u)int64 arrays. Arrays no longer store 1 rank.
0.4.0	Compressed the Table of Contents Sections
0.3.0	This version has known issues and is ot to be used
0.2.0	Added support for prepend and append fields in a ListOp
0.1.0	Fixed structure layout issues encountered on Windows systems
0.0.0	Initial Release

16.3.8.3 Bootstrap

Following the header is a bootstrap that gives the location of the table of contents, and a section of data reserved for future use.

Description	Size	Value
Table of Contents Offset that must point to a byte offset from the start of the file	8 bytes	Unsigned 64 bit integer
Reserved	8 bytes	

The reserved bytes are recommended to be zero value bytes, but are not required to be so currently. They may be used in future versions of the specification for specific semantic meanings.

16.3.8.4 Table of Contents

The table of contents is found at the byte offset provided in the bootstrap. The Table of Contents must be at the end of the file.

Description	Size	Value
Number of Sections	8 bytes	Unsigned 64 bit integer
Repeating Section Data	32 bytes per Section	See Table below

The Table of Contents represents Section listings. Each listing is defined by the following pattern. Individual listings follow each other consecutively.

Description	Size	Value
Name	16 bytes	ASCII string. Null Terminated
Start Offset representing the start of the section from the start of the file	8 bytes	Unsigned 64 bit integer
Size representing the number of bytes to read	8 bytes	Unsigned 64 bit integer

The number of sections is variable, but the following are the standard sections used to populate the data in a common crate layer. Each section is optional and can be elided if they don't have data to represent as long as they're not also required by another section.

16.3.8.4.1 Tokens Section

The tokens section defines all the tokens within the file in their compressed form. It is called `TOKENS` in the Table of Contents.

The section starts with the number of tokens, defined by an unsigned 64-bit integer. Following this, are two unsigned 64-bit integers that represent the uncompressed and compressed size of the data.

Finally, the data is stored in an LZ4 compressed series of bytes, whose length is defined by the compressed size that was previously specified.

Once uncompressed, the data represents a series of Tokens that must each be null terminated.

The tokens array must include an item at its first index (0) that is not referenced by the other sections. This can be any string value that you choose to author, but it is recommended to keep it short.

Each token represents a UTF-8 encoded string.

Description	Size	Value
Number of Tokens	8 bytes	Unsigned 64 bit integer
Uncompressed size as a number of bytes	8 bytes	Unsigned 64 bit integer
Compressed size as a number of bytes	8 bytes	Unsigned 64 bit integer
Data	Represented by the compressed size	LZ4 Compressed byte array, null delimited and null terminated

16.3.8.4.2 Strings Section

The strings section is an array of Indices into the array of tokens read in the Tokens section. It is called `STRINGS` in the table of contents. It must be processed after reading the [Tokens](#)

section.

The first element of the section is the number of indices to be read. Following that is the contiguous array of Index values of this count.

Description	Size	Value
Number of Indices	8 bytes	Unsigned 64 bit integer
Data	4 bytes per Index , times the number of indices	

16.3.8.4.3 Fields Section

The fields section stores a relationship between token names and a Value Representation as defined in the Value Representations section of the document below.

The fields section is called FIELDS in the table of contents. It must be processed after reading the [Tokens section](#).

The section starts with an unsigned 64-bit integer representing the number of fields that are encoded.

Following this is a Compressed Integer Array representing an array of Indices of tokens from the [Tokens section](#).

After this, is an unsigned 64-bit integer representing the size of the compressed value representations.

Finally, this is followed by the compressed data of that size. This data is compressed using the LZ4 Compression schema above. This data represents the various value representations.

Description	Size	Value
Number of Fields	8 bytes	Unsigned 64 bit integer
Compressed Indices	Defined by the Compressed Integer Array algorithm	Compressed Integer Array
Data Size	8 bytes	Unsigned 64-bit integer
Value Data	Defined by the Data size	

16.3.8.4.4 Field Sets Section

The Field Sets section stores a grouping of fields that are presented together. It is called FIELDSETS in the Table of Contents.

It must be read after the [Fields section](#).

The section starts with an unsigned 64-bit integer representing the number of Indices.

Field Sets are stored in a flat array of Indices into the Fields section, where groups are terminated by a maximum-value Index (4,294,967,295 or -1 if using unsigned values).

Description	Size	Value
Number of Fields	8 bytes	Unsigned 64 bit integer
Data	As defined by Compressed Integer Arrays algorithm	Array of Indices

16.3.8.4.5 Paths Section

The Paths section stores a list of paths used in the file. It is called PATHS in the Table of Contents. This section depends on the [Tokens section](#) being parsed first.

Each path is stored as a deconstructed set of three arrays with the same length of items. The resulting paths are stored out of order so that they may be constructed from the results of this array.

The section starts with an unsigned 64-bit integer representing the total number of paths, followed by three compressed integer arrays of equal length as described in the subsections below.

The data layout of this section is as follows.

Description	Size	Value
Number of Paths	8 bytes	Unsigned 64 bit integer
Path Indices		Array of Indices
Element Token Indices		Array of signed 32-bit integers
Jumps		Array of signed 32-bit integers

16.3.8.4.5.1 Path Indices Array

Following this is the Path Indices array. It is a compressed integer array of indices that represents the Index of the path.

16.3.8.4.5.2 Element Token Index Array

Next is the Element Token Index array, represented as a compressed integer array of signed 32-bit integers.

The absolute value of the token_index is a lookup into the Tokens index.

The resulting token is added to a parent path using either a prim path or property path delimiter as per this table

The Element Token Index must never point to the zero index, as different languages have difficulties delineating between positive and negative zero. The root element must be a forward slash.

Type	Description
Positive	Represents a prim path (/ delimiter)
Negative	Represents a property path (. delimiter)

16.3.8.4.5.3 Jump Array

Finally, there is the Jumps array, represented as a compressed integer array of signed 32-bit integers.

The construction of the hierarchy of this path is defined by the Jump. Jump values have significance as described in this table below. This allows for optimised storage of wide hierarchies which are more common than very deep hierarchies.

A positive jump represents where to jump to next in the Path Indices and Element Token Index array to construct the paths of these path additions.

Value	Description
>0	An offset to a sibling relative to the current index
0	This index only has siblings.
-1	The index only has a child
-2	This index is a leaf

16.3.8.4.5.4 Path Construction Algorithm

Paths are constructed from the indices using a recursive algorithm based on the three arrays defined in this section.

The array is constructed out of order, so it is important to pre-allocate an output array when starting this algorithm,

The steps to replicate the algorithm are as follows.

Start with a current index (X) of 0, to represent the first iteration. Start with an empty Parent Path.

Loop through the following steps, and increment the current index at the end of each loop. Exit the loop if the current path is a leaf path.

1. If the parent path is empty, set the parent path and current path to / to represent the absolute root path. If it is not empty then
 1. Get the paths index from the Path Indices array at X.
 2. Get the Element Token Index at X to find what Token to lookup
 3. Use the absolute value of the element token index to look up the Tokens array for the new token to add.

4. Add the token to the parent path using the path or property delimiter according to the positive or negative value of the element token index.
2. Store the resulting path in your output array at the path index
3. Get the Jump value from the Jump array at X. This value is used to check if the path has siblings or children.
4. If the jump value is a leaf node, stop here.
5. If the jump value indicates it has siblings (0 or higher), then run this algorithm from the start with the starting index as the current X plus the jump value.
6. Set the parent path to the value in the output array at X

16.3.8.4.6 Specs Section

The Specs section combines data from the previous sections to create a PrimSpec. The section is called SPECS in the Table of Contents.

This section depends on the [Paths section](#) being parsed first.

A spec represents the final elements of the document, such that each element in the document is a combination of:

1. A Path
2. A Form
3. A set of fields that apply to this path

The section starts with an unsigned 64-bit integer representing the number of specs.

Following this are three compressed integer arrays:

1. Path indices consisting of Indices into the Path section
2. Field Set indices consisting of Indices into the Field Sets section
3. Forms which are a series of unsigned 32-bit integers corresponding to the Spec's Form identifier

Field Set Indices must point to the index of the start of a group within the flat Field Set section.

Description	Size	Value
Number of Specs	8 bytes	Unsigned 64 bit integer
Path Indices	As defined by Compressed Integer Arrays algorithm	Array of Indices
Field Set Indices	As defined by Compressed Integer Arrays algorithm	Array of Indices
Forms	As defined by Compressed Integer Arrays algorithm	Array of unsigned 32-bit integers

The normative spec forms are described in the following table.

Value	Form	Description
0	Unknown	An unknown Form
1	Attribute	Attributes under a Prim Spec
6	Prim	A Prim Specifier
7	Layer	Holder of layer specific fields
8	Relationship	A Relationship Description
10	Variant	A specific variant within a Variant Set
11	VariantSet	A group of variants

Unknown spec forms may be processed by the implementation, but are inert.

An additional set of spec forms are considered non-normative, and are present for compatibility purposes with older crate files. Binary file format parsers should be able to consume the specifier but may ignore any semantics around it. These compatibility spec forms are described in the following table.

Value	Form	Description
2	Connection	Connections between Rel attributes
3	Expression	Scripted Expressions or named Plugin expressions. Internal to Pixar.
4	Mapper	Used to modify the value flowing through a connection. Internal to Pixar.
5	MapperArg	Arguments for a Mapper. Internal to Pixar.
9	RelationshipTarget	A target for a relationship

16.3.9 Value Representations

The Crate format stores value representations as data blobs that are read on demand. This allows large USD scenes to be read quickly by deferring data reads.

Value representations are stored as unsigned 64-bit integers with specific significant bytes.

The first 6 bytes are reserved for the Payload of the value representation.

The penultimate byte represents the value representations type, enumerated in the [Value Types](#) type table.

The last byte acts as bit flags to represent characteristics of the type.

Description	Size	Value
Payload	6 bytes	
Type	1 byte	The type this value represents.
Bit Flags	1 byte	Flags describing this value representation

The possible bit masks are

Description	Value
Value is an Array	The last bit
Value is Inlined	The second last bit
Value is Compressed	The third last bit

The Array and Compressed bit flags may be combined.

16.3.9.1 Inlined Value Representation

A Value that has the Inlined bit flag set, is stored directly in the Payload of the Value representation. There are a few rules to the encoding of inlined values:

- Only the first 4 bytes of the payload may be used to encode data.
- Inlined values cannot be arrays or be compressed.
- If the payload is zero/null, then the value is implicitly an empty constructed or zero valued form of the value type.
- If the value(s) type is a floating precision data type, and if the type cannot fit in the 4 byte limit, it is only inlined if the value can be encoded as a signed 8-bit integer. e.g. Vec2h can fit in 4 bytes so stores the half-precision floating values directly in the inlined value, but Vec3h cannot fit, so can only inline values that are representable as signed 8-bit integers.
- The payload is parsed differently based on the dimensionality of the type.

Description	Value
Single dimension (e.g Integers, Floats)	Directly cast to the type
Two Dimensions (e.g Vec2i)	Stored as contiguous values that convert to the given type
Three Dimensions (e.g Matrix3D)	Stored as the top-left to bottom-right diagonal values of the given type E.g 1 - - - 1 - - - 1

The following Value types must always be inlined:

- bool
- uchar
- int
- uint
- half
- float
- Variability
- ValueBlock
- AnimationBlock

- string (by index)
- token (by index)
- ObjectPath (by index)
- asset (by index)

The following value types may be inlined by a crate writer, and must be supported by readers:

- double (if the value is identical when cast to float)
- int64 (if the value is in range of int32)
- uint64 (if the value is in range of uint32)
- matrixXd (if diagonal and diagonal elements are exactly represented when cast to int8_t)
- dictionary (if empty)

16.3.9.2 Offset Value Representation

If a value representation is neither an Array nor Inlined, it is an offset representation. The payload acts as an offset of bytes from the beginning of the document to data that can be parsed into the specific value type being represented.

16.3.9.3 Array Value Representations

Array value representations have the array bit flag set. They may optionally also have the compressed bit flag set.

If the payload is zero, the array is considered empty. Otherwise, the value will represent the byte offset from the start of the document.

The data at the offset will start with an unsigned 64-bit integer, representing the number of elements in the array, followed by the data.

An array value can be parsed in multiple ways depending on the payload and the compression bit.

Situation	Value
Payload is 0	The array is empty
Uncompressed	Data is stored as a contiguous array of the type specified
Compressed	Array compression will vary based on the type of the value representation.

Arrays that are 16 bytes or smaller should not be compressed.

Only arrays that have simple data type representations per element may be compressed. e.g. Integers and Indices may be compressed, but Quaternions and Matrices may not.

16.3.9.3.1 Compressed Integral Arrays

Compressed Integral arrays are stored using the compressed integer array algorithm.

16.3.9.3.2 Compressed Floating Point Arrays

Compressed float arrays start with an 8-bit character representing the array encoding scheme.

Integer arrays are used when all the floats can be represented as integers.

Encoding Character	Value
i	Uses the compressed integer array algorithm
t	Uses a lookup table (described below)

The size of the lookup table (LUT) is an unsigned 32-bit integer.

This is followed by a contiguous Value array of the given type.

Finally, a compressed array of integers representing the Indices.

The Indices point to values in the Value array. This allows for the de-duplication of values.

Description	Size	Value
LUT Count	4 bytes	Unsigned 32 bit integer
LUT Values	LUT Count * size of type	
LUT Indices	LUT Count * Index size	

16.3.10 Value Types

16.3.10.1 Type Table

A table enumerating the USD value types, with their respective IDs. These map to the types in the [Foundational Data Types](#) section.

ID	Name	Supports Array
1	bool	Yes
2	uchar	Yes
3	int	Yes
4	uint	Yes
5	int64	Yes
6	uint64	Yes
7	half	Yes
8	float	Yes
9	double	Yes
10	string	Yes
11	token	Yes
12	asset	Yes
13	matrix2d	Yes
14	matrix3d	Yes

ID	Name	Supports Array
15	matrix4d	Yes
16	quatd	Yes
17	quarf	Yes
18	quath	Yes
19	double2	Yes
20	float2	Yes
21	half2	Yes
22	int2	Yes
23	double3	Yes
24	float3	Yes
25	half3	Yes
26	int3	Yes
27	double4	Yes
28	float4	Yes
29	half4	Yes
30	int4	Yes
31	dictionary	No
32	TokenListOp	No
33	StringListOp	No
34	PathListOp	No
35	ReferenceListOp	No
36	IntListOp	No
37	Int64ListOp	No
38	UIntListOp	No
39	UInt64ListOp	No
40	PathVector	No
41	TokenVector	No
42	Specifier	No
43	Permission	No
44	Variability	No
45	VariantSelectionMap	No
46	TimeSamples	No
47	Payload	No
48	DoubleVector	No
49	LayerOffsetVector	No
50	StringVector	No
51	ValueBlock	No
52	Value	No
53	UnregisteredValue	No
54	UnregisteredValueOp	No
55	PayloadListOp	No
56	TimeCode	Yes

ID	Name	Supports Array
57	PathExpression	Yes
58	Relocates	No
59	Splines	No

Specific types are called out below in regard to how to parse them.

16.3.10.2 **bool**

A boolean binary value, where any non-zero value is considered True. Booleans are 8-bit values.

16.3.10.3 **uchar**

An unsigned, 8-bit character.

16.3.10.4 **int**

A signed 32-bit integer.

16.3.10.5 **uint**

An unsigned 32-bit integer.

16.3.10.6 **int64**

A signed 64-bit integer. When inlined it is represented as an Int.

16.3.10.7 **uint64**

An unsigned 64-bit integer. When inlined, it is represented as a UInt

16.3.10.8 **half**

A 16-bit floating point number.

Since many languages lack a standard representation of a half, USD uses the [Imath based Half implementation](#).

This Half implementations follows the conventions of other floating point numbers but uses reduced bit range for the exponent and significand.

The first bit is the sign-bit, denoting whether it is positive or negative.

The next 5-bits are the exponent.

The remaining 10-bits are the significand

Description	Size
Sign	1 bit
Exponent	5 bits
Significand	10 bits

Description	Size
-------------	------

As with other floating point numerical types, there are standard conventions per the IEEE-754 specification. They are described here for convenience.

If the Sign bit is 0, it is positive. If it is 1, it is negative.

If the exponent is 31, the number is considered as either Infinity or Not a Number (NaN).

If the exponent is between 1 and 30, the half is a normalized number.

If the exponent is 0 but the significand is not 0, it is a denormalized number.

If the exponent and significant are both zero, then the resulting value is zero.

16.3.10.9 float

A 32-bit float.

16.3.10.10 double

A 64-bit floating point number. When inlined , it is represented as a float.

16.3.10.11 string

Strings are stored in the file as an Index to a token in the Strings section.

16.3.10.12 token

Tokens are stored in the file as an Index to a token in the Token section.

16.3.10.13 asset

Assets are stored in the file as an Index to the String section. However, when inlined, they point to an index in the Token section.

16.3.10.14 Path Expression

Path Expression support was introduced with Crate version 0.10.0 .

Path Expressions have the same representation as an AssetPath.

16.3.10.15 Relocates

Relocates are stored as a dictionary of key, value pairs of AssetPaths.

Relocates support was introduced with Crate version 0.11.0 .

The first element is a 64-bit unsigned integer to designate the size of the dictionary.

Following this are a series of Indices into the Paths section, alternating between related keys and values.

Description	Size	Value
Element Count	8 bytes	Unsigned 64-bit Integer
Key	4 bytes	Path Index
Value	4 bytes	Path Index

16.3.10.16 ValueBlock

ValueBlocks a sentinel type that indicates that a value is authored but is specifically set to no value. As such, ValueBlocks have no data bytes.

Value Blocks may not be stored inside an array.

16.3.10.17 Value

A Value is an indirect pointer to another value somewhere else in the file.

It is represented by an unsigned 64-bit integer offset which points to a Value Representation at the given offset from the start of the file.

It is important to guard against recursion here so that pointers don't create an infinite loop. If recursion is detected, an empty value can be returned.

A Value may not be stored inside an array

16.3.10.18 UnregisteredValue

A representation of metadata fields that are not registered with the system.

An unregistered value starts with a signed 64-bit integer as an offset from its current position. This new position points to a Value.

The Value can only point to the following other value types:

- Strings
- Dictionaries
- UnregisteredValueListOps

Unregistered Values may not be stored within an array.

16.3.10.19 Dictionary

A Dictionary is a key:value map of data, where the Key must be a Token.

The Value Representation starts with an unsigned 64-bit integer representing the number of elements in the dictionary.

Following the count are each key:value stored consecutively.

Keys are stored as an Index to the Token section of the document.

Values are stored as a 64-bit signed integer representing the offset from the current

Dictionaries may not be stored in an array.

An empty dictionary is represented implicitly as one that is inlined.

Description	Size	Value
Element Count	8 bytes	Unsigned 64-bit Integer
Key	4 bytes	Index
Value Offset	8 bytes	Signed 64-bit Integer

16.3.10.20 Layer Offset

Layer Offsets are represented by two 8-byte Doubles.

The first Double represents the time offset to be used.

The second Double represents the Scale.

Description	Size	Value
Time Offset	8 bytes	Double
Scale	8 bytes	Double

16.3.10.21 References and Payloads

References are stored as four consecutive fields. Payloads are a form of reference but are otherwise identical.

The first field is the asset layer path, represented by an Index to an asset path in the Strings section. An empty string represents an internal reference.

The second field is an Index into the Paths section, representing the Prim to use. If no prim is specified, then this uses the default prim per the composition engine.

The third field is 16-bytes for a Layer Offset.

The fourth and final field is a Dictionary for Custom Data , as defined by the Dictionary type below. This only exists for references and not for Payloads.

References may not be stored inside an Array.

Description	Size	Value
Asset Layer Path	4 bytes	Index
Prim Path Index	4 bytes	Index
Layer Offset	16 bytes	Layer Offset
Custom Data	8 bytes	Dictionary (only for References)

16.3.10.22 Quaternions

Quaternions are represented by four contiguous elements of the given Quaternion type. The first three elements make up the imaginary coefficients. The last element is the real coefficient.

When displayed to a user (especially when converted to text), the real coefficient must be displayed in front of the imaginary coefficients.

Quaternions may have the following base floating precision types.

Quaternion Type	Base Type
quatd	Double
quarf	Float
quath	Half

16.3.10.23 Vectors (Mathematical)

Vectors are dimensioned types a fixed length, contiguous array of a given type.

Vectors as a type represent a mathematical vector, and should not be confused with the Vector Array type described further down in this section.

The following dimensioned types exist

Dimensioned Type	Base Type	Count
double2	double	2
float2	float	2
half2	half	2
int2	int	2
double3	double	3
float3	float	3
half3	half	3
int3	int	3
double4	double	4
float4	float	4
half4	half	4
int4	int	4

16.3.10.24 Matrix

Matrix types are a 2-dimensional array of doubles, of the same size (N) in both directions.

Matrix types are defined in a contiguous row-major order, such that `matrix[i][j]` refers to row `i` and column `j`.

Identity Matrix values have all elements zeroed out, except from the top left (0,0) to bottom right (N,N) which are set to 1.

Matrices are represented as dimensioned types, the following of which are available

Dimensioned Type	Dimension	Identity Matrix
matrix2d	2 x 2	1 0 0 1
matrix3d	3 x 3	1 0 0 0 1 0 0 0 1
matrix4d	4 x 4	1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1

16.3.10.25 List Operations

List Operations, or ListOps, are a Value Representation of an operation that edits a list of items.

List Operations may not be stored in an array themselves, or inlined.

List Operations allow for the following operations:

- Adding items
- Adding Explicit Items
- Deleting items
- Reordering items
- Appending items
- Prepending Items
- Making Items explicit

The List Operation starts with an 8-bit bitmask header that determines the operation being performed.

Description	Size	Value
Header	1 byte	Bitmask

The Header byte stores the following bit masks from least significant to most significant. If only Make Explicit is true, then the list operation can be treated as being cleared with no elements.

Shorthand	Description	Bytemask
Make Explicit	Removes all items and changes the list to be explicit	0x1 (1 << 0)
Add Explicit Items	Fills the list with the items included in this ListOp	0x2 (1 << 1)
Add Items	Adds the items included in this list op	0x4 (1 << 2)
Delete Items	Deletes the items specified in this list op	0x8 (1 << 3)
Reorder Items	Reorder the list based on the item order within this ListOp	0x10 (1 << 4)
Prepend Items	Prepends items from this ListOp to the start	0x20 (1 << 5)
Append Items	Append items from this ListOp to the end	0x40 (1 << 6)

Following the header is a series of arrays depending on the bit masks above in the following order:

1. Add Explicit Items
2. Add Items
3. Prepend Items
4. Append Items
5. Delete Items
6. Reorder Items

Each array starts with an unsigned 64-bit integer representing the number of elements. This is followed by a contiguous array of the ListOps core type, with that number of elements.

Description	Size	Value
Element Count	8 bytes	64-bit unsigned integer
Data	Element Count * Core Data Type	Uncompressed elements

The following core types are supported within a List Operation Type

List Operation Type	Base Type
TokenListOp	Token Index
StringListOp	String Index
PathListOp	Path
ReferenceListOp	Reference
PayloadListOp	Payload
IntListOp	int
Int64ListOp	int64
UIntListOp	uint
UInt64ListOp	uint64
UnregisteredValueListOp	UnregisteredValue

Note: Some list operations specified above in the binary format specify compatibility with older OpenUSD binary files. Specifically, Add and Reorder are meant for compatibility and have no normative semantic.

16.3.10.26 Vectors (Arrays)

Vectors are variable length arrays of a given type stored contiguously. They are not to be confused with the mathematical vectors described further up in this section. Their naming reflects the variable-length container types in programming languages like C++.

Vectors always start with an unsigned 64-bit integer reflecting the number of elements it contains, and is followed by a contiguous array of that data type.

Arrays of Vector Arrays are not permitted.

Description	Size	Value
Element Count	8 bytes	64-bit unsigned integer
Data	Element Count * Core Data Type	Uncompressed elements

The following data types are valid within a Vector

Vector Type	Base Type
PathVector	Index into Path section
TokenVector	Token Index
DoubleVector	double
LayerOffsetVector	Layer Offset
StringVector	String Index

16.3.10.27 Specifier

Specifiers represent the composition specifier of a given Prim. They are represented as a signed 32-bit integer with the following valid values

Spec Type	Description	Value
Def	Concretely defining specifier	0
Over	Non-defining specifier	1
Class	Abstractly defining specifier	2

16.3.10.28 Permission

Permissions are a deprecated field within USD documents, that originally represented how a layer might refer or express an opinion about a prim.

They are documented here for posterity, but are otherwise not required to be supported. They must not be authored into new USD files.

Permissions are represented as a signed 32-bit integer, and cannot be stored in an array.

Permission Type	Value
Public	0
Private	1

16.3.10.29 Variability

Variability is represented by a signed 32-bit integer with the following valid values, and may not be stored in an array.

Refer to the [Variability](#) section for information about the values.

Variability Type	Value
Varying	0
Uniform	1

16.3.10.30 Variant Selection Map

Variant Selection Maps are a key:value map. The key is the name of a Variant Set, and the value is a variant within that Variant Set.

Multiple entries within the map can share the same key, as a given Variant Set may have many variants.

A Variant Selection Map starts with an unsigned 64-bit integer representing the number of elements.

Following this is a contiguous series of String Index, alternating key and value.

Variant Selection Maps may not be stored in an array.

Description	Size	Value
Element Count	8 bytes	64-bit unsigned integer
Key	4 bytes	Index
Value	4 bytes	Index

16.3.10.31 Timesamples

Timesamples store a series of time varying Value Representations.

Timesamples cannot be stored in an array or be inlined.

At the payloads offset, is a 64-bit signed integer that represents the offset from this current location.

Description	Size	Value
Offset to Timecodes	8 bytes	Signed 64-bit Integer

At this new offset, is a Value Representation of the samples stored as a DoubleVector.

Following this, is a signed 64-bit integer that points to an offset from this point to the values array.

Description	Size	Value
Offset to Values	8 bytes	Signed 64-bit Integer

The values array starts with an unsigned 64-bit integer that represents the number of elements, and an array of Value Representations.

The TimeCodes and Values are mapped to each other by index.

Description	Size	Value
Number of Values	8 bytes	Signed 64-bit Integer
Array of Values	8 bytes * Number of Values	

16.3.10.32 TimeCode

TimeCode shares the same representation as a Double.

TimeCode support was introduced with Crate version 0.9.0 .

16.3.10.33 Splines

Splines are defined as a series of bytes for the spline data, and an alternating series of TimeCodes and Dictionary data.

Spline support was introduced with Crate version 0.12.0 .

Splines are always stored at the offset value and may not be inlined.

At the offset, we find an unsigned 64-bit integer that tells us how many bytes of spline data will be there. Following this is a series of bytes of that count.

Description	Size	Value
Byte Count	8 bytes	64-bit unsigned integer
Bytes	Byte Count	A byte array

After this is another unsigned 64-bit integer that tells us how many custom data items there are. Following this is an alternating couple of this count, mapping and consisting of TimeCodes and a Dictionary.

Description	Size	Value
Custom Data count	8 bytes	64-bit unsigned integer
TimeCode	8 bytes	TimeCode double
Dictionary	Refer to Dictionary	Dictionary

Once both are read, the dictionary bytes may be parsed using the dictionary encoding algorithm.

The first byte is a flag byte with the following bit usage from lowest to highest.

Description	Size and Position
Version	4 bits (byte & 0x0F)
Data Type	2 bits ((byte & 0x30) » 4)

Description	Size and Position
Timed Value	1 bit (byte & 0x40)
Curve Type	1 bit ((byte & 0x80) » 7)

Currently, we document version 1 of the splines format.

The Data Type is mapped with

Value	Description
0	Unspecified (treated as double)
1	Double
2	Float
3	Half

Timed value is a boolean.

Curve Type is a boolean where 0/false indicates that it is a Bézier curve, and 1/true indicates that it is a Hermite Curve.

Following this flag byte is another flag byte with the following flags

Description	Size and Position
Pre Extrapolation Mode	3 bits (byte & 0x07)
Post Extrapolation Mode	3 bits ((byte & 0x18) » 3)
Is Looping	1 bit (byte & 0x40)

The extrapolation modes are mapped with the following values

Value	Description
0	Block
1	Held
2	Linear
3	Sloped
4	Loop Repeat
5	Loop Reset
6	Loop Oscillate

If either the pre-extrapolation or post-extrapolation are set to Sloped, the slope data is stored after this as a double for each.

Description	Size	Value
Pre Extrapolation Slope	8 bytes if Pre Extrapolation mode is Slope	Double
Post Extrapolation Slope	8 bytes if Post Extrapolation mode is Slope	Double

If the Is Looping bit is set, the following loop parameters are encoded next

Description	Size	Value
Proto Start	8 bytes	TimeCode
Proto End	8 bytes	TimeCode
Number of Pre Loops	4 bytes	Positive, unsigned 32-bit integer with a maximum value of a signed 32-bit integer
Number of Post Loops	4 bytes	Positive, unsigned 32-bit integer with a maximum value of a signed 32-bit integer
Value Offset	8 bytes	TimeCode

16.3.10.33.1 Knots

If the Data Type is not Unspecified, it is followed by bytes that encode the knots of the spline.

To start is an unsigned 32-bit integer that tells us how many knots are encoded.

Following that is the knot information

Description	Size	Value
Knot Count	4 bytes	Unsigned 32 bit integer

Each knot is consecutively encoded

Description	Size	Value
Flag Byte	1 byte	Single bit-array byte

The flag bits are encoded like so from lowest to highest

Description	Size and Position
Dual Valued	1 bit (byte & 0x01)
Interpolation Mode	2 bits ((byte & 0x06) » 1)
Curve Type	1 bit ((byte & 0x08) » 3)
Pre Tangent is in Maya Form	1 bit (byte & 0x10)
Post Tangent is in Maya Form	1 bit (byte & 0x20)

Following the flag byte is a sequential set of data

Description	Size	Value
Time	8 bytes	TimeCode
Value	Variable	The data type of the splines
Dual Value	Variable	The data type of the splines. Only encoded if Dual Valued flag bit is true.
Pre Tangent Width	8 bytes	Double if curve type for knot is not Hermite
Post Tangent Width	8 bytes	Double if curve type for knot is not Hermite
Pre Tangent Slope	Variable	The data type of the splines
Post Tangent Slope	Variable	The data type of the splines

16.4 Package

USD Packages store USD files and other required media within a specialized zip compression

16.4.1 ZIP

USD Packages are encoded according to the ZIP standard with some caveats.

USDZ normatively references the Zip File Format Specification version 6.3.9 of PKWARE® Inc. USDZ are conforming Zip files as specified by the document presented here <https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT>.

A breakdown of the PKZip format is provided by the James Madison University at <https://users.cs.jmu.edu/buchhofp/forensics/formats/pkzip.html>.

16.4.1.1 ZIP Features

USD Packages must be stored without compression and must be stored without encryption. USD Packages must be 32 bit ZIP files and not 64 bit.

16.4.1.2 Contents

A USD Package must use the first file within the Zip file as its root layer to be read.

The file must be the first entry in the Zip Central Directory, as described by the PKZip specification.

Additionally, the file must be the first file provided from the beginning of the Zip file to allow for efficient loading when reading a zipfile straight-ahead without jumping to the central directory record.

16.4.1.3 Data Layout

USD Packages must be aligned to 64 byte blocks, such that every file header starts at a multiple of 64 bytes for efficient reading.

16.4.1.4 End of Central Directory record Restrictions

To provide efficient lookup of the zip contents, USDZ restricts the use of some fields in the End of Central Directory record. This particularly also speeds up the look-up along a network or remote location where latency can be very high. By not having to search for the End of Central Directory record, USDZ can be efficiently read across high latency boundaries.

Additionally, the End of Central Directory record must be at the very end of the zip file, with no padding following it.

The final bytes of the file must be as such

Description	Size	Value
Signature	4 bytes	The standard PKZIP signature of 50 4B 05 06
Disk Number	2 bytes	A value of zero as USDZ does not support multi disk zips
Central Directory	2 bytes	Also, a value of zero
Disk Number		
Central Disk	2 bytes	The number of entries in the central directory of this file
Entries		
Total Disk Entries	2 bytes	The number of entries in the central directory of this file, as there may only be one central directory
Central Directory	4 bytes	The size of the central directory in bytes
Size		
Central Directory	4 bytes	The offset of the Central directory from the start of the
Offset		file
Comment Length	2 bytes	This must be zero as no comment is allowed

The comment length must be the final set of bytes in the file with nothing after it. This restriction may be loosened in future versions of the specification.

16.4.1.5 File Types

USDZ files are recommended to only include the following file types/extensions for maximum portability:

- .usd , .usda , .usdc
- .png , .jpg , .jpeg , .exr , .avif
- .m4a , .mp3 , .wav

You may include other file types, but this will risk the compatibility of the USDZ package with other runtime that may not support those formats.

16.4.2 Validation and Out of Spec USDZ files

Implementations of this specification are not required to validate a USDZ file when reading from it.

Implementations may choose to support out of spec USDZ files, or they may choose to reject it.

For example, if a USDZ file does have a comment, or doesn't align to the data layout restrictions, it may be still easily read by a range of zip libraries.

However, out of spec usdz files are not guaranteed to be portable across implementations, and implementations must adhere to the USDZ specification when writing the file.

16.5 References

1. Ford, Bryan (January 2004) *"Parsing Expression Grammars: A Recognition Based Syntactic Foundation"*. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. pp. 111-122. <https://dl.acm.org/doi/10.1145/964001.964011>

17 Closing

This Specification describes the core of USD as of the time of its writing. OpenUSD continues to evolve, and new additions to OpenUSD will aim to be captured in future Specification versions.

17.1 Acknowledgements

This Specification is the work of many companies and individuals as part of the Core Specification Working Group under the [Alliance for OpenUSD](#).